PARTITIONING GRAPH DATABASES VIA ACCESS PATTERNS

by

Volkan Tüfekçi

B.S., Computer Science Engineering, Marmara University, 2008

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2013

PARTITIONING GRAPH DATABASES VIA ACCESS PATTERNS

APPROVED BY:

Prof. Can Özturan (Thesis Supervisor)	....
Assoc. Prof. Haluk Bingöl	....
Assoc. Prof. Turgay Altılar	....

DATE OF APPROVAL: 5.22.2013

ABSTRACT

PARTITIONING GRAPH DATABASES VIA ACCESS PATTERNS

With the emergence of large scale social networks such as Twitter, Facebook, Linkedin and Google+ the growing trend of big data become much clear. In addition to storing this highly connected big data, an efficient mechanism for processing this data is also needed. The inadequacy of traditional solutions such as relational database management systems for processing highly connected data caused the people head toward graph databases. Graph databases are the natural fit for connected data with their underlying data structure model depending on graphs. They are able to handle up to billions of nodes and relationships on a single machine but the high growing rate of social data pushes their limits. In this study, we evaluate partitioning graph databases in order to increase throughput of a graph database system. For this purpose we designed and implemented a framework that both partitions a graph database and provides a fully functional distributed graph database system. Comparing to previous studies we have concentrated on access pattern based partitioning. Within our experiments access pattern based partitioning outperformed unbiased partitioning that only depends on static structure of the graph. We have evaluated our results on real world datasets of Erdös Webgraph Project and Pokec social network.

ÖZET

ÇİZGE VERİ TABANLARINI ERİŞİM ÖRÜNTÜLERİ İLE BÖLÜMLEME

Twitter, Facebook, Linkedin gibi büyük ölçekli sosyal ağların ortaya çıkmasıyla büyük verinin artan büyüme eğilimi daha da belirgin hale geldi. Bu durum, yoğun şekilde bağlı bu büyük verinin saklanmasına ek olarak etkili bir mekanizmayla işlenmesi gereksinimini doğurdu. İlişkisel veri tabanı yönetimi sistemleri gibi geleneksel çözümlerin yetersiz kalması insanların çizge veritabanlarına yönelmesine sebep oldu. Çizge veritabanları, veri yapısı modellerinin çizgeleri temel alması nedeniyle bağlı veriler için doğal bir çözüm olmaktadır. Milyarlarca düğüm ve ilişkiyi tek bir makinede işleyebilmelerine rağmen sosyal verinin artan büyüme hızı limitlerini zorlamaktadır. Bu çalışmada bir çizge veri tabanı sisteminin işlem hacmini arttırabilmek için çizge veri tabanlarının bölümlenmesini değerlendirmeyi amaçladık. Bu doğrultuda hem çizge veri tabanını bölümlendiren hem de dağıtık bir çizge veri tabanı sistemi sunan bir yapıyı tasarladık ve gerçekledik. Önceki çalışmalardan farklı olarak erişim örüntülerine dayanan bir bölümlendirme üzerinde yoğunlaştık. Denemelerimiz esnasında erişim örüntülü bölümlendirme, sadece çizgenin yapısına dayanarak bölümlendirme yapan yöntemi geride bıraktı. Değerlendirmelerimiz için Erdös Projesi ve Pokec sosyal ağ verilerini kullandık.

TABLE OF CONTENTS

ABSTRACT iii
ÖZET iv
LIST OF FIGURES vii
LIST OF SYMBOLS/ABBREVIATIONS ix
1. INTRODUCTION 1
2. BACKGROUND 3
2.1. NoSQL 3
2.2. Graphs & Graph Databases 3
2.2.1. Neo4j 5
2.2.2. Graph Database v.s Relational Database 6
2.2.3. Graph Database Applications 8
2.2.4. Graph Traversal Languages 9
2.2.4.1. Traversal Framework API 9
2.2.4.2. Cypher 12
2.2.4.3. Gremlin 13
2.3. Scale-free Networks 14
2.4. Amazon Web Services 16
2.5. Related Work 16
3. GRAPH DATABASE PARTITIONING FRAMEWORK VIA ACCESS PATTERNS 19
3.1. Graph Partitioning Methodologies 19
3.1.1. Blind Partitioning 19
3.1.2. Access Pattern Weighted Edge Partitioning 21
3.1.3. Access Pattern Hash Partitioning 21
3.2. Graph Partitioning Sub-steps 22
3.2.1. Feeding Redis 23
3.2.2. Access Pattern Collecting 25
3.2.2.1. APC for APWEP 27
3.2.2.2. APC for APHP 27

3.2.3. Pre-partitioning Phase 27
3.2.4. Partitioning 29
3.2.5. Preparing Neo4j Instances 30
3.3. Runtime Architecture 32
3.3.1. Client 33
3.3.2. Redis Server 34
3.3.3. Result Database 35
3.3.4. Neo4j Instances 37
4. TESTS AND EVALUATION 41
4.1. Datasets 41
4.1.1. Erdös Webgraph Dataset 41
4.1.2. Pokec Social Network Dataset 42
4.2. Tests 43
4.2.1. Query (Traversal) Performance Testing 43
4.2.2. Load Testing 45
5. CONCLUSION 51
5.1. Contribution 51
5.2. Future Work 52
REFERENCES 53

LIST OF FIGURES

Figure 2.1. NoSQL data models with data size vs. data complexity aspects [1]. 4

Figure 2.2. Basic labeled property graph terminology. 6

Figure 2.3. Entitiy relationship diagram of user and friendship tables in SQL. 10

Figure 2.4. SQL query for finding friends of a user. 10

Figure 2.5. SQL query for finding FOF of a user. 11

Figure 2.6. Traversal for FOF of a user in Traversal Framework API. 11

Figure 2.7. A 2 depth FOF Cypher query. 14

Figure 2.8. A 2 depth FOF Gremlin query. 14

Figure 3.1. Overall schema of BP process. 20

Figure 3.2. Query dispatching at runtime. 22

Figure 3.3. Sample graph file in Scotch input format with no weight. 28

Figure 3.4. A sample graph with edge weights and corresponding graph file in Scotch input format. 29

Figure 3.5. Runtime architecture of BP and APWEP. 32

Figure 3.6. Runtime architecture of APHP. 33

Figure 3.7. JSON sent from client to the system in order to start a query. . . 34

Figure 3.8. Unpartitioned sample graph with 11 nodes. 38

Figure 3.9. Partitioned sample graph with 11 nodes. 38

Figure 3.10. Sample JSON sent during delegation. 40

Figure 4.1. Degree distribution of Erdös Webgraph Dataset [2]. 42

Figure 4.2. BP 2 depth query performance results with extra traffic. 44

Figure 4.3. BP 3 depth query performance results with extra traffic. 45

Figure 4.4. APWEP 2 depth query performance results with extra traffic. . . 46

Figure 4.5. APWEP 3 depth query performance results with extra traffic. . . 47

Figure 4.6. 2 depth load test results. 48

Figure 4.7. 3 depth load test results. 49

Figure 4.8. APHP load distribution. 49

Figure 4.9. BP load distribution. 50

Figure 4.10. APWEP load distribution. 50

LIST OF SYMBOLS/ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation, Durability
AGPL	Affero General Public Licence
APBP	Access Pattern Based Partitioning
API	Application Programming Interface
AWS	Amazon Web Services
BP	Blind Partitioning
CPU	Central Processing Unit
CSV	Comma Separated Values
EC2	Elastic Computing Cloud
FOF	Friends of Friends
GID	Global ID
JAR	Java Archive File
JSON	Javascript Object Notation
REST	Representational State Transfer
S3	Simple Storage Service
URL	Uniform Resource Locator

1. INTRODUCTION

With the emergence of large scale social networks such as Twitter, Facebook, Linkedin and Google+ the growing trend of highly connected data become much clear. Social data produced by these social networks has a very high growing rate and as more and more people get involved in this connected social world, growth rate does not seem likely to lose momentum in the near future. High penetration of mobile devices in the market also supports this growth by providing interaction from anywhere at anytime which means nothing but more social data. Even an outdated information about Facebook's data storage [3] from 2010 tells us how big is this data with 21 PB of storage within 2,000 machines.

The amount of highly connected social data produced by large scale social networks made companies behind them confront big data problems. Problems like horizontal scalability for higher throughput, efficient storage for durability and efficient data processing for retrieving useful information from data. These problems pushed the limits of the dominant player of data storage systems (relational database systems) since 1970's and gave birth to the NoSQL era.

NoSQL databases are specialized players of the big data world and they are good at horizontal scaling because of being designed as distributed stores from the beginning. They handle up to billions of records in petabyte size. Graph databases are one of these NoSQL databases and they are specialized in handling data with graph storage. They have similar characteristics as the other NoSQL databases but most of them lack an efficient partitioning mechanism in order to handle more data for more concurrent clients via distributing its data and load.

The problem with partitioning graph databases is an inherent one related to graph partitioning problem. Graph partitioning has the complexity of NP-hard as graphs are structures designed to be connected instead of being partitioned. Solutions are based on approximations or heuristics.

The main contribution of this work is a graph database partitioning framework with a novel approach for partitioning graph databases via access patterns. Our work is based on the idea proposed in Averbuch & Neumann's work [4]. Three partitioning methodologies, which of two utilize access patterns, are implemented and compared to each other. A distributed graph database runtime architecture that runs over Amazon Web Services (AWS) [5] is also designed and developed.

The remainder of this thesis is organized as follows. Chapter 2 gives all related background information about graph partitioning, graph databases, traversal frameworks and scale-free networks. Also, it explains relevant academic publications and similar studies. The proposed graph database partitioning system is described in Chapter 3 with the details of partitioning methodologies used in Section 3.1 and runtime architecture in Section 3.3. The tests and comparison of partitioning methodologies with the definition of datasets used are given in Chapter 4. Finally, in Chapter 5 we provide an overall conclusion and discussion of ideas for future works.

2. BACKGROUND

2.1. NoSQL

First usage of the NoSQL term is in 1998 by Carlo Strozzi to name his relational database that did not follow the SQL standard intentionally [6]. But the meaning we use for NoSQL is shaped around a conference about non-relational databases organized by Johan Oskarsson in 2009 [7]. This time the term is referred as "Not only SQL" but according to Strozzi it should have been called as "NoREL" as the intention is departing from SQL's relational world.

NoSQL world is an example of objection to "one size fits all" philosophy. For different use cases and for different needs different NoSQL databases are developed which could be categorized into four groups according to their data model:

- Key-value stores: Keeps data in key-value pairs (Redis, BerkeleyDB...)
- Column-oriented: Keeps data in a one big table with millions of rows and billions of columns (Bigtable, Cassandra)
- Document models: Keeps data in versioned document structures which consists of key value pairs (MongoDB, CouchDB)
- Graph databases: Data is modeled with graphs (Neo4j, Infinitegraph)

Figure 2.1 taken from Emil Eifrem's "NoSQL overview and intro to graph databases with Neo4j" talk [1] in 2010, summarizes the comparison of NoSQL database models. Comparing to other NoSQL models Graph databases deal with smaller sized data but conversely the complexity of the data is higher.

2.2. Graphs & Graph Databases

As their name implies, graphs are the basis of the data structure used in graph databases to model data. In mathematics graph is a representation of objects and links

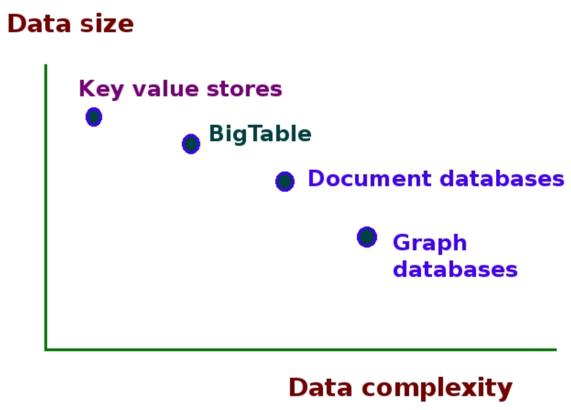


Figure 2.1. NoSQL data models with data size vs. data complexity aspects [1].

between them with dots and lines. These dots and lines are called *vertices* and *edges* respectively. So, the graph is an ordered pair with a notation $G = (V, E)$. Vertices are also called nodes and edges are called arcs or relationships.

Despite being a simple structure graphs are able to model vast amount of real world data. One of the reasons behind this is we are living in a world of relationships; all the objects are related to other objects in one sort of another. Another reason is that our brain models real world like graphs and even our brain structure is like a graph with neurons and their connections between them resemble a graph structure [8].

The most common graph type is simple graph which is comprised of set of vertices and undirected edges that connect exactly two different vertices. Even though simple graphs are powerful enough to model data, they are enriched with different capabilities for different use cases. For example, properties are added to vertices to keep related data within vertex, direction is added to edges to discriminate different flows. In Table 2.1 a short summary of graph types is given. More detail is given in Rodriguez & Neubauer's article [9].

We will be talking about property graphs that support "directed", "labeled" and "attributed" graph types. Property graphs allow defining properties (attributes) for both nodes and relationships in order to fulfill data model requirements. This basic

Table 2.1. Summary of type of graphs.

Graph Type	Description
Simple	An undirected edge connects two unique vertices
Multi	More than one edge may connect two same vertices
Weighted	Edges have capacity or cost
Vertex-labeled	Vertices have identifiers
Vertex-attributed	Vertices have meta-data
Edge-labeled	Edges are denoted by their type
Directed	Edges have directions
Semantic	Meaning of the graph is machine understandable
Hypergraph	An edge may connect more than on vertex

graph terminology is shown in Figure 2.2.

Graph databases are storage engines that support one or more of the graph types and persist data in native graph format. Furthermore, accessing underlying data with querying options is provided in a pleasant and performant way. Neo4j is one of the prominent graph databases and for the remainder of this study we will refer to Neo4j when we use the term graph database.

2.2.1. Neo4j

Neo4j is the leading graph database solution developed by Neo Technology. It is open-source with Affero General Public Licence (AGPLv3) [10] and also commercially licensed for professional support option.

Neo4j uses *property graphs* as data model and provides graph native disk-based persistence. Providing ACID (Atomicity, Consistency, Isolation, Durability) compliant transactions makes Neo4j a reliable storage system.

Neo4j is developed with Java programming language and has a well documented,

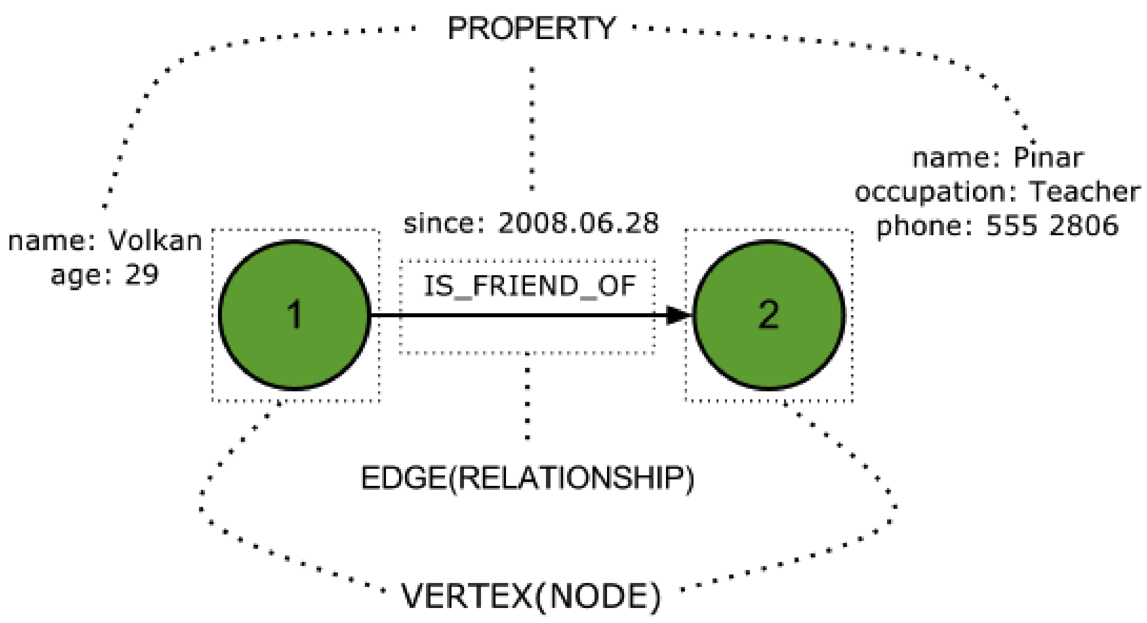


Figure 2.2. Basic labeled property graph terminology.

object oriented API. It has a mature code base with a vibrant community.

With a small footprint Neo4j could be used embedded in Java code with a single Java Archive File (JAR) import or it could be run as a server with REST oriented data access option.

2.2.2. Graph Database v.s Relational Database

Relational database management systems have been the dominant player of data storage solutions for many decades. They emerged in 1970's and became the de-facto standard of both in industry and academia. They range from small sized databases such as SQLite, HSQLDB to large scale commercial databases such as Oracle and SQL Server. Their high penetration in the market give birth to an eco-system with wide variety of tools such as query browsers with graphical user interfaces, query profilers and optimizers and native clients in almost every programming language.

As they have been used for many many real world cases, relational databases are also able to model graph data in relational model such like NoSQL databases [11]. But this is kind of a situation when you have an hammer, you begin to see everything like nail. It is undesirable to use a relational database in an unnatural domain like graphs.

Relational databases are designed for tabular data with fixed schema requirements. Their real power comes into play when the queries are related to set theory and deals with the overall data depending strongly on indexes. Comparatively, graph databases use graph models underneath, which provides local queries without the need of an index. All the entities are adjacent to each other so edges are directly accessible from their vertices and vertices are directly accessible from their entities. This index--free model makes graph databases surpass any other storage mechanism for holding highly connected data because by this way regardless of the total dataset size, time required for traversing the graph stays constant. This is similar to counting people 5 seats away from you in a stadium. The time required is independent from the total number of people in the stadium. Graph databases still use indexes for traversing the graph but just for once in order to locate the beginning of the query, start node. On the other hand, relational databases need to perform index related operations for fetching edges of a vertex or vertices of an edge.

Relationships are first class citizens of graph databases. Unlike relational databases or other kind of NoSQL solutions, there is no need to infer relationships searching foreign-keys or jumping from table to table. Relationships are explicit and mandatory.

Another advantage of graph databases over relational databases is schema-free data modeling. Relational databases needs to use strict schemas for defining the structure of data. This schema evolves until it meets the requirements of the use case and every change means recreating related tables. On the other hand, graph databases are schema-free models which allow defining new kinds of relationships or properties for nodes in runtime.

Object-relational impedance mismatch [12] is also a disadvantage of relational databases. Object oriented programming languages use object model which does not align completely with the relational model. However, graph model could be directly modeled in object oriented world.

As a result, there is no silver bullet and for highly connected graph like structures, relational databases are far from being a silver bullet when we have the option of choosing graph databases.

2.2.3. Graph Database Applications

Graph databases are well suited in situations where data is highly connected and queries to this data are formed as local traversals through the portions of the graph. Social networks, web graphs, telecommunication networks are good examples of highly connected data. Searching, ranking, recommending and scoring are local traversal operations on these data.

Graph databases are getting more and more popular in social network operations. For example Twitter's friend recommendation system is a 2 hop traversal which is queried as "Find people that follows the people that I follow". Similarly Facebook's recently announced new feature, Graph Search [13], is also a graph traversal. "People who like cycling and lives in Izmir" could be translated as a query like "Find people that are connected to both Cycling with LIKES relationship and Izmir with LIVES_IN relationship". This way underlying data would be more understandable as the data and data model match.

As with social networks, recommendation engines are one other area that graph databases shine. Similar to social networks, recommendation engines strongly depend on relationships between entities special to a domain. Most of the time this relationship occurs as a preference of a user for an item such as book, movie or song. The rating attached to a preference could be easily modeled as a property attached to a relationship with a property graph model supporting graph database.

Geospatial is the original use case of graphs [14]. Euler gave a solution of Seven

Bridges of Konigsberg problem which later came to form the basis of graph theory. Geographically separated data that forms a network is geospatial data. Power grids, transportation networks and telecommunication networks are examples of geospatial data which could be modeled in graph databases.

With its highly interconnected structure Bioinformatics is also a suitable area for graph databases. Actually, Bio4j [15] is a well known graph database application developed just for this purpose.

2.2.4. Graph Traversal Languages

"Querying tables" in SQL world is translated into "Traversing nodes and relationships" in graph database domain. This difference comes from the reason that records are stored as rows in tables and relationships between tables are covered with foreign keys in SQL databases. Conversely, in graph databases records are stored in vertices and relationships between these vertices are stored in edges which corresponds to nodes and relationships in Neo4j terminology. It is important to emphasize that relationships are first class citizens of graph database world. They have labels, directions and properties.

There are three frameworks for traversing operations in Neo4j. These are:

- Traversal Framework API
- Cypher
- Gremlin

and they are explained in following sections.

2.2.4.1. Traversal Framework API.

Traversal Framework API is the programmatic interface of traversing facility provided by Neo4j graph database. It is coded in Java and as it is a direct connection to the underlying structure it is the most performant one.

The natural structure of a social network is much closer to graph model, so it is possible but both inefficient and hard to model these kind of networks in relational databases. A simple friendship network modeled in SQL is shown in Figure 2.3. The entity-relationship diagram tells us that there are two tables, USER for user records and FRIENDSHIP for friendship relation records with indexes on the ID columns for improving query performance. And there is a relationship between USER and FRIENDSHIP tables but that's the whole information we may get about the relationships of the domain. Despite the name of the model is "relational model" relationships are poorly modeled. Additional information related to relationships such as label, direction or properties could not be stored on the relations of relational models.

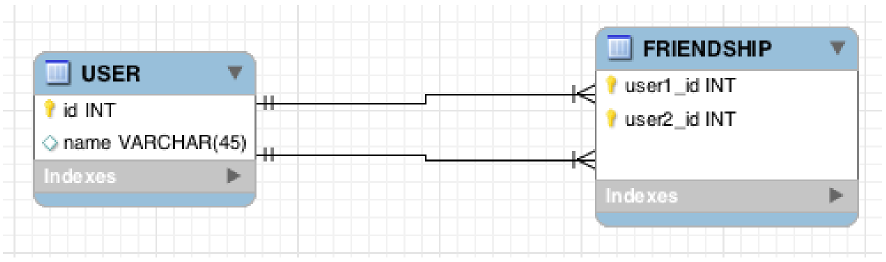
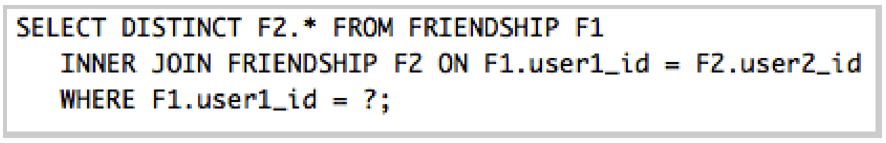


Figure 2.3. Entitiy relationship diagram of user and friendship tables in SQL.

In order to find friend of a given user, required SQL query is given in Figure 2.4. It is a simple query with one join. Figure 2.5 shows the SQL query for a 2 depth friends of friends (FOF) query which is common in online social networks. Within this query the picture of 3, 4 or 5 depth FOF queries become clear that we would need more and more joins which will get more complex in each level of an increase in depth.

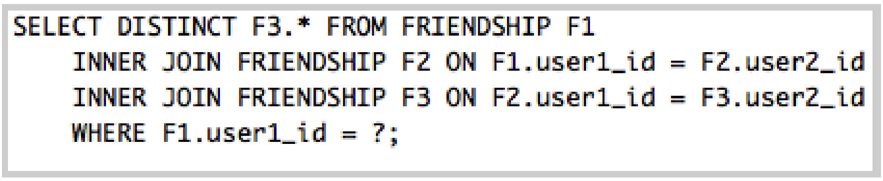


```
SELECT DISTINCT F2.* FROM FRIENDSHIP F1
   INNER JOIN FRIENDSHIP F2 ON F1.user1_id = F2.user2_id
   WHERE F1.user1_id = ?;
```

Figure 2.4. SQL query for finding friends of a user.

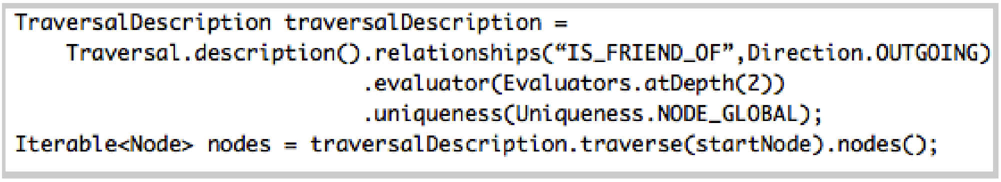
For a comparison, required traversal code in order to find FOF of a user is given in Figure 2.6. There are 2 main steps: Constructing traversal description and iterating.

Traversal description is the definition of how the graph will be traversed. As the



```
SELECT DISTINCT F3.* FROM FRIENDSHIP F1
    INNER JOIN FRIENDSHIP F2 ON F1.user1_id = F2.user2_id
    INNER JOIN FRIENDSHIP F3 ON F2.user1_id = F3.user2_id
    WHERE F1.user1_id = ?;
```

Figure 2.5. SQL query for finding FOF of a user.



```
TraversalDescription traversalDescription =
    Traversal.description().relationships(“IS_FRIEND_OF”,Direction.OUTGOING)
                           .evaluator(Evaluators.atDepth(2))
                           .uniqueness(Uniqueness.NODE_GLOBAL);
Iterable<Node> nodes = traversalDescription.traverse(startNode).nodes();
```

Figure 2.6. Traversal for FOF of a user in Traversal Framework API.

relationships in Neo4j has labels, we first constrain our traversal to friendship relationship with using “IS_FRIEND_OF” label for relationships. Next, we define direction of our relationship as “OUTGOING” because we are considering friends of friends. Reciprocally, if we were dealing with “whose friend is given user” then we would use “INCOMING” as the direction of our relationship. Finally, we define how many depths should our traverse go on via “atDepth (2)”. Otherwise the traverser may traverse till the end of graph. An additional constraint is given as “Uniqueness.NODE_GLOBAL” which will constrain our traverser not to visit a node twice. Uniqueness could be different in different situations and Traversal Framework API provides other uniqueness constraints such as “RELATIONSHIP_GLOBAL, NODE_PATH, NODE_RECENT, RELATIONSHIP_PATH, RELATIONSHIP_RECENT”.

Iterating is the step that the traversing happens. The only needed parameter is the start node of traversal which could be found out with a single index look up. Traverser traverses the graph starting from the given node with constraints given with traversal description.

Besides being more comprehensible comparing to SQL query, increasing the depth of FOF of a user query is simply as increasing the “atDepth” value of the traversal description for Traversal Framework API. So, Traversal Framework API stays intelligible independent from the depth.

Unnatural and complicated structure of SQL query for social networks is not the only downside of relational databases. The performance of these queries is also inefficient with magnitudes of order comparing to graph traversals. In Table 2.2 a comparison of run times for querying a sample social network with 1.000.000 records with 50 friends for each performed by Partner & Vukotic and shared in their book [16] is shown.

Table 2.2. Performance comparison of SQL vs. Traversal Framework API [16].

Depth	RDBMS Execution Time (s)	Neo4j Execution Time (s)	Records Returned
2	0.016	0.01	∼2,500
3	30,267	0.168	∼110,000
4	1,543,505	1.359	∼600,000
5	Unfinished	2.132	∼800,000

The difference is negligible for depth 2 but it becomes significant from depth 3 and upwards. The reason behind this could be explained by the inefficiency of SQL queries for nested joins. SQL performs cartesian products of records on tables that it joins. This equals to $50.000.000^5$ rows for a 5 depth query. Additionally, most of these rows will need to be discarded to filter friends. Both of these operations requires significant amount of time and computation power. However, graph databases are born to deal with deep relationships. Traversals are local queries that deals with only related nodes to the queried node without the need to include all records into play like the joins of SQL databases.

2.2.4.2. Cypher. Cypher [17] is the default query language of Neo4j and does not require coding in Java or any other language. It is kind of an abstraction over the Traversal Framework and comes as a built-in plugin. Neo Technology is the developer and maintainer of Cypher.

Cypher is designed by veteran users of SQL so it is no surprise that Cypher is very similar to SQL as they have mutual keywords which operate similarly such as WHERE, ORDER, LIMIT and so on. This similarity was intentional with the purpose of easing the procedure of transition from relational database systems to Neo4j for *data related people*. The *data related people* are database administrators, analysts, data scientists or even programmers. SQL is widely known within these people so it would not be so hard to switch their queries to Neo4j traversals from the point of syntax aspect.

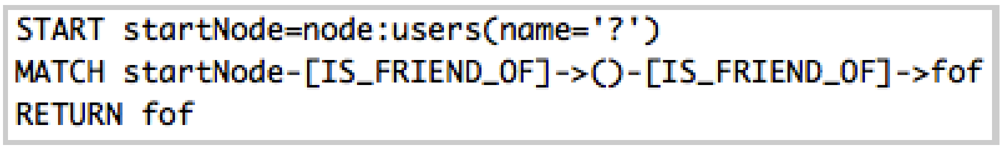
Cypher specifies "what to fetch" more than "how to fetch". A Cypher is query is converted into Traversal Framework API under the hood, so it may work less performant in some situations. But for most of the use cases it is desirable to use Cypher for quick access to data with a succinct syntax and with the ease of using a web browser.

Despite being a traversal language Cypher is able to modify the graph database with create, update or delete operations on nodes and relationships.

Cypher could be called from Java code or the console of Neo4j Server accessed by a web browser. Cypher could also be called from any programming language by embedding a Cypher query in JSON and sending it to the Representational State Transfer (REST) end-point of Neo4j Server.

The FOF query introduced in Section 2.2.4.1 is given in Figure 2.7 as a Cypher query. On the first line starting node of the traversal is found via an index look-up. In our case it is assumed that there is an index with name "users" and holds *name-node* pairs as key-values. "?" mark should be replaced by the name of the user we are querying for. On the second line, *MATCH* clause matches the relationships with given order, type and direction. Also it constraints the depth of traversal to 2 via defining "IS_FRIEND_OF" relationships twice. Finally, third line returns nodes matched in the second line.

2.2.4.3. Gremlin. Gremlin [18] is another popular graph traversal language. It is developed by Thinkerpop and works on any graph database that implements BluePrints

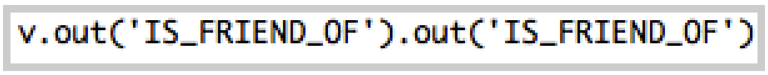


```
START startNode=node:users(name='?')
MATCH startNode-[IS_FRIEND_OF]->()-[IS_FRIEND_OF]->fof
RETURN fof
```

Figure 2.7. A 2 depth FOF Cypher query.

interface including Neo4j. It is built on top of Groovy dynamic programming language [19] as a domain specific language [20] so Gremlin may use all the functionality provided by Groovy. Gremlin could be executed as embedded in any JVM language code, or called from REST endpoint similar to Cypher.

Gremlin equivalent of FOF query is given in Figure 2.8. It is just a simple line with two keywords: *v* denotes the vertices and *out* denotes an outgoing relationship. *out* is actually called *gremlin step*. Similarly there are other gremlin steps such as *in, both* for returning nodes and *outE, inE, bothE* for returning relationships.



```
v.out('IS_FRIEND_OF').out('IS_FRIEND_OF')
```

Figure 2.8. A 2 depth FOF Gremlin query.

2.3. Scale-free Networks

Many of the real world complex networks fall in the category of scale-free networks. Computer networks such as internet and web graph of WWW, proteins and genes interactions, citation networks, nervous systems, airline networks, power grid networks and online social networks are conceptually similar to each other as they are well known examples of scale-free networks.

A network is accepted as scale-free when the probability of a vertex to have a relationship with k other vertices decays as a power law independent from the network size given in [21] as follows:

$$P(k) \sim k^{\gamma} \tag{2.1}$$

and γ value changes typically in the range of $2 < \gamma < 3$. Power law could be briefly explained as where large sized events in a network are rare but opposingly small sized events are common [22]. There are just few users with millions of followers like Barack Obama but on the other hand there are millions of users that have very few followers as average degree is below than 100 [23]. There are few web sites with great number of visitors but very large number of sites with very few or no visitors. Most of the world's income is shared by small amount of people comparing to rest of the world with modest income.

Earlier random network models such as Erdös and Renyi (ER) model are insufficient for modeling scale-free networks because of not considering the key properties of power law; growth and preferential attachment. Real networks does not have fixed number of vertices (N) like ER model assume its network, instead; (N) grows continuously via addition of new vertices connected to the already existing vertices. Preferential attachment is known as "rich get richer". A new vertex has higher probability to connect to an existing vertex with a large degree of connections instead of connecting to a random vertex uniformly. This may be observed from Twitter that a celebrity's follower count's growth rate is larger than an ordinary user.

Scale-free networks has hubs within the network which have number of degrees that far more exceeds the average degree of the network. These hubs have some special properties that varies from network to network such as a celebrity or a politician in a social network, an airport that connects many flights in a airline network, a web portal that contains many links to other web sites in a web graph network. Hubs connect smaller dense graphs of the network called communities. Removing hubs makes the network disconnected.

If the degree distribution of a network fits a power law distribution then it may be considered as a small-world network. Small world network's diameter grows proportional to the logarithm of network's size.

2.4. Amazon Web Services

Amazon Web Services is one of the prominent cloud computing architectures with a huge range of complete services. AWS behaves as Platform-as-a-Service (Beanstalk, web container), Software-as-a-Service (Simple Storage Service, highly durable, redundant data storage service) and Infrastructure-as-a-Service (Elastic Computing Cloud, virtual machine running Linux or Windows) at the same time.

The purpose of AWS is providing a high available system with on-demand horizontal and vertical scaling options. The system works as "pay as you go" fashion.

We used AWS as the test-bed of our work. We run both partitioning framework and distributed graph database architecture on AWS. Partitioner machine, result database, Redis server, partitioned Neo4j instances and finally clients are all run on Elastic Computing Cloud (EC2) instances. For storing common files such as "graph.db" directories related to partitioned graph and configuration files Simple Storage Service (S3) is used.

We developed Bash scripts for orchestrating EC2 instance provisioning operations and automating procedures run at machine boot-up process which is explained in Section 3.2.5.

2.5. Related Work

The studies related to graph partitioning and graph processing started at 1970's [24, 25]. Graph databases appeared in 1990's [26] but the research around hypertext made the subject almost disappear [27]. Although graph partitioning and graph data models have been around for many years there are very few studies related to graph database partitioning.

Pregel [28] is a framework developed by Google for processing large graphs that have up to billions of vertices and trillions of edges. It is designed by Bulk Synchronous Parallel [29] model. Pregel's results are impressive but it does not address graph partitioning directly as it uses random partitioning. And also Pregel runs in main memory which limits size of the graph within the boundaries of involving machines' total memory.

Averbuch & Neumann [4] evaluated graph partitioning algorithms for partitioning Neo4j graph database. Three algorithms were studied and one of them, DIDIC, was implemented. The authors evaluated the algorithms over three datasets: Twitter, transportation of Romania and a synthetically generated one. With the lack of real access logs of the datasets, they defined access patterns for the datasets and based their results on these access patterns over the partitioned data sets. They developed a prototype via extending Neo4j GraphDatabaseService API in order to support same operations provided by Neo4j for a seamless partitioned graph database service from the point of user. Shadow vertex model, which is similar to our proposed solution, was used but the prototype was not physically distributed. Also, as the prototype performed poorly an emulator is used for the rest of the study. It is claimed by the authors that the performance penalty occurred because of the increased software stack. Their evaluation showed that modularity optimizing graph partitioning algorithm was a good fit for load balance across partitions and less network traffic.

Chairunnada, Forsyth, Daudjee performed a study [30] similar to ours. They have examined three methods for partitioning a graph database and discussed two models for handling inter-partition edge problem. They selected "Vertex Partitioning" method and implemented a partitioned graph database called PNeo4j via extending Neo4j. They tested their implementation with simple scenarios and demonstrated that a partitioned graph database could be developed without introducing significant amount of latency, excluding network latency, as opposed to the results achieved by Averbuch & Neumann [4]. The model chosen for inter-partition edges was Dangling Edge Model which uses ghost edges instead of ghost vertices. A real edge is the one kept within the same partition of its start node and forwards all the requests to the remote partition. It is claimed in the study that via eliminating ghost vertices storage cost is downsized to one ghost edge but in our opinion this should be further explored because according to the Neo4j documentation [31] average memory requirement for a node (9 bytes) is approximately 4 times smaller than average memory requirement for a relationship (33 bytes). This means that using ghost edges may cause more storage consumption in such cases that extra ghost vertex count is 3 or lesser times smaller than ghost edge count. Another problem with Dangling Edge Model is that Neo4j guarantees [32] to have valid start and end nodes for a relationship which means extensive implementation is needed in order to alleviate this problem but no further detail is given. They performed their experiments on a single machine with each partition running on a different port and their testing scenario was traversing over an edge for 1000000 times. Finally, they stated that their implementation lacks shipping traversal processing to remote partitions as only the partition contacted with the client is able to traverse and they added that they expect implementation of shipping traversal processing to remote partitions would improve performance.

Ho, Wu, Liu [33] also studied evaluation of an efficient distributed graph database architecture for processing large scale online networks. They implemented an architecture that involves a distributed graph data processing system and a distributed graph data storage. They used GoldenOrb [34] for graph data processing with a modification to make it connect to Neo4j as data store. Their choice of partitioning method was using METIS [35] in favor of reducing cut-edges comparing to hash partitioning. The datasets used were relations of people with each other from online social networks such as Youtube, Orkut, Flickr and LiveJournal. Max value propagation, single source shortest path, bipartite matching, influence spreading, page rank and random walk is used as benchmarks for comparing traffic generated by hash partitioning and METIS partitioning. Maximum value computation was another performance comparison between Neo4j and Hadoop as a graph data storage system. They indicated that by the help of indexing and fast graph traversal capabilities Neo4j was a definite winner over Hadoop within the context of execution time.

3. GRAPH DATABASE PARTITIONING FRAMEWORK VIA ACCESS PATTERNS

In this chapter we explain the graph database partitioning framework developed in detail. The system is composed of two main parts: Graph Partitioner and Runtime Architecture.

Graph Partitioner is the first deliverable of our study. As its name implies, graph partitioner takes an unpartitioned graph data and partitions it into desired number of partitions with selected partitioning methodology. Resulting partitioned data is the input of Runtime Architecture.

Runtime Architecture is the second deliverable of our study which is a distributed graph database architecture that connects the parts of the system such as Neo4j instances, Redis server and result database. Runtime Architecture is supported with an intelligent client that is able to infer which instance to send its query via communicating with Redis server.

We have shared both of the products as Github repositories [36, 37].

3.1. Graph Partitioning Methodologies

We have implemented three different graph partitioning methodologies: Blind Partitioning (BP), Access Pattern Weighted Edge Partitioning (APWEP) and finally Access Pattern Hash Partitioning (APHP). Comparison and evaluation of these methodologies are given at Chapter 4.

3.1.1. Blind Partitioning

Blind Partitioning is partitioning the graph as is, without using any heuristics based on access patterns. Partitioning library, Scotch [38], partitions the graph with the purpose of minimizing edge cuts. As it is developed for thread-process distributing for multi-core central processing units (CPU), Scotch's default tendency is balancing the load between partitions via distributing vertices to partitions with equal number of vertices on each partition. But in our case we have tuned the imbalance ratio of Scotch to 0.9 in order to minimize hops during traversals as it will compensate the imbalanced number of vertices on each partition.

In Blind Partitioning, a Neo4j graph database or a comma-separated values (CSV) file is taken as an input and loaded into Redis if needed. After these initial operations an input file for Scotch is prepared based solely on edges. Next, partitioning is performed by Scotch and finally, partitioned Neo4j instances are created from the output of Scotch library. This process is displayed in Figure 3.1.

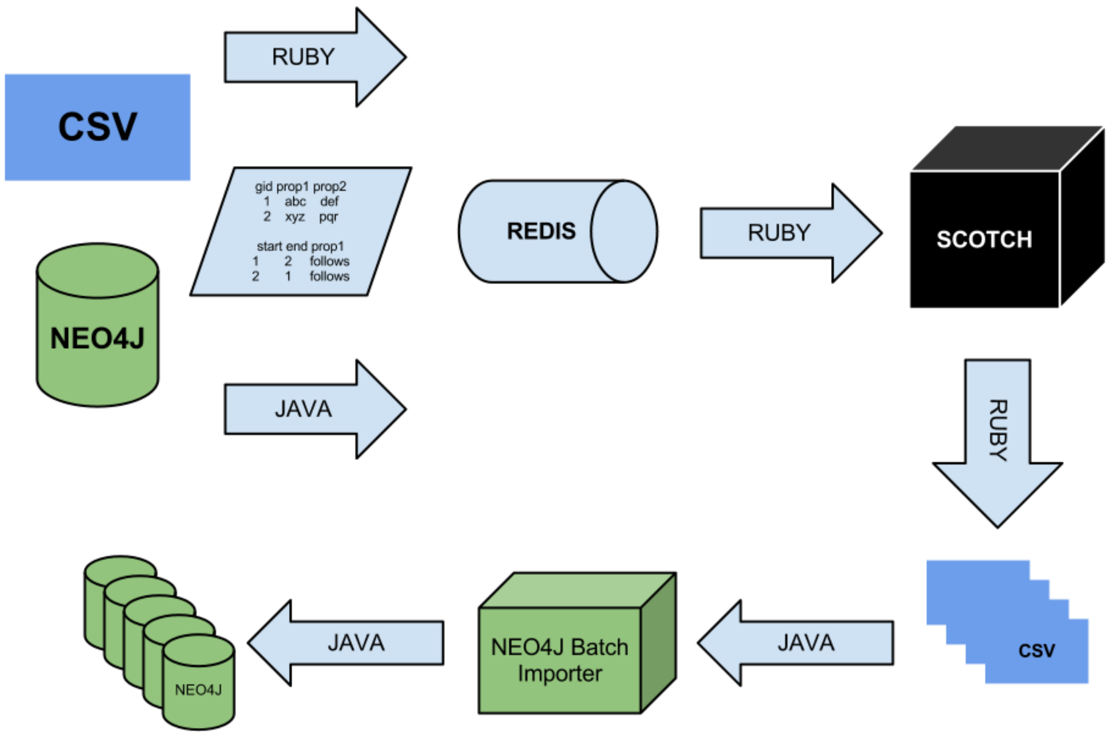


Figure 3.1. Overall schema of BP process.

3.1.2. Access Pattern Weighted Edge Partitioning

Access Pattern Weighted Edge Partitioning (APWEP) is similar to BP but this time edges have weights depending on the access patterns collected.

Every edge starts with a weight of 1 as a default value. The weight is increased whenever an edge is visited during traversals. Obviously the more visited an edge, the more weight it will have. As a result of many many traversals some edges become more important than others as their weight indicates that it should not be cut by a partitioning operation. Because, most of the traffic flows over these edges with high weights. Cutting them means so many hops to another instance during traversals will need to be performed and this would lead to the most undesired situation that we call *extra traffic*.

Similar to BP, APWEP uses Scotch for partitioning the graph data.

3.1.3. Access Pattern Hash Partitioning

Access Pattern Hash Partitioning (APHP) is not a truly partitioning methodology like the ones we introduced before; in fact it may be seen as a sub-graph separation method. The major difference is same node may occur in different partitions as a *real node*.

APHP utilizes both access patterns and hash partitioning. The decision of locating starting node of a query is given by hash partitioning which is simply calculating the modulus of a start node ID based on desired partition count. And access patterns are used to collect nodes during the traversals. These collected nodes are located into the same partition within their start node. As a result, the partitions are formed around access patterns and related nodes.

One of the Neo4j instances holds the same graph before partitioning, it contains all the nodes and relationships so it is called *master instance*. Any query that could not be fulfilled by a partition will end up at this instance. The other instances of the system could be thought as a cache for the master instance; whenever a miss occurs for a query then it is forwarded to master instance. The runtime query dispatching algorithm is given below:

```
   Input: query: A query in json, startNodeID: GID of start node of query,
          partitionCount: Total Partition Count
 1 begin
 2     $queryHash \leftarrow$ hash($query$)
 3     if doesExistInRedis($queryHash$) then
 4         $partitionNumber \leftarrow startNodeID$ mod $partitionCount$
 5     end
 6     else
 7         $partitionNumber \leftarrow masterInstance$
 8     end
 9     $partitionURL \leftarrow$ fetchURLOfPartitionFromRedis($partitionNumber$)
10     sendQueryToPartition($partitionURL$)
11     traverse()
12     uploadResultToDB()
13 end
```

Figure 3.2. Query dispatching at runtime.

APHP assumes that updates to nodes and relationships occur at the master instance are propagated to other instances. The design of such a system is beyond the scope of this project.

3.2. Graph Partitioning Sub-steps

In the Graph Partitioning Methodologies section (Section 3.1) we explained the methodologies we used. In this section we will explain the details of the graph partitioning methodologies as sub-steps. Some of the sub-steps are common to all methodologies and some of them are used exclusively. The sub-steps are as in the following:

- Feeding Redis
- Access Pattern Collecting
- Pre partitioning
- Partitioning
- Preparing Neo4j instances

3.2.1. Feeding Redis

The input of this step is nodes and relationships, plus their properties. The source of the input could be CSV files or a Neo4j instance. The output of the step is a filled up Redis server.

This step is optional, so if the Neo4j instance that needs to be partitioned is synced with Redis server already; it could be skipped. Its usage is common for all three partitioning methodologies.

Redis is an open source, in-memory NoSQL solution within the class of Key-Value Store written in ANSI C. It is referred as an advanced data structure server with its built-in data types such as strings, hashes, lists and sets for its keys. Redis keeps all its data in memory but at the same time it supports persistence via transferring the data from memory to disk asynchronously. It supports replication with master-slave configuration. High Availability and Clustering are on-going efforts. There are variety of clients for Redis written in all the major programming languages such as Java, Ruby, C++.

The purpose behind integrating Redis to our partitioning system was improving the time spent during the data processing in order to prepare the input of Scotch partitioning library and Batch Importer. We have chosen Redis for being open source, fast and persistent at the same time and having stable clients written in Ruby and Java.

We store the same data both in Neo4j and Redis, but Redis stores the data as keys and values which makes it far more faster than Neo4j when fetching and inserting large amounts of data during Batch Importing process (Section 3.2.5).

The most important part of storing data in Redis is the selection of *key structure* which is similar to schema organization process of relational databases. Our way of selecting the key structure is shown in Table 3.1. The nodes are mapped with just one key pattern denoted as *node:X* and X is the GID of a node. Node representation of the sample graph with 11 nodes (given in Figure 3.8) is given in Table 3.2. Nodes in the sample graph have only 1 property, called *name*, but the structure is open to grow with additional properties and values.

In order to map relationships two groups of patterns are used per relation. First one is denoted as *rel:R* and used for storing properties of relationship (R is the ID of relationship). Second pattern is denoted as *in:G* or *out:G* according to the direction of the relationship and G is GID of related node. The purpose of this separated key structure for relationships is gaining performance improvement because otherwise relationship hash maps would need to be stored in a complex structure with hash maps inside hash maps which would increase processing cost. Relationship representation of sample graph for nodes 1, 2 and 3 is given in Table 3.3. In & out relationships representation of node 2 is given in Table 3.4.

Table 3.1. Key structure of Redis.

Key	Sample Value	Explanation
node:G	{name:Volkan}, {age:30}	G: GID of a node
rel:R	{Start:s},{End:e},{Type:t}	R: Rel. ID, s&e: Start&End node GIDs
out:G	$\{r{:}o\}$	G: Node GID, r: Rel. ID, o: Other node GID
in:G	$\{r{:}o\}$	G: Node GID, r: Rel. ID, o: Other node GID

Table 3.2. Node representation of sample graph in Redis.

Redis Key	Property	Value
node:1	name	a
node:2	name	b
node:3	name	c
node:4	name	d
node:5	name	e
node:6	name	f
node:7	name	g
node:8	name	h
node:9	name	i
node:10	name	j
node:11	name	k

3.2.2. Access Pattern Collecting

Access Pattern Collecting (APC) is the crucial part of our proposed method for partitioning graph database. Partitioning with Scotch is based on the output of this phase.

The relationships in a graph database reflect the underlying structure of the data. This structure is an important source for acquiring further knowledge about the data but individually it is not enough for spotting hot points within the context of queries. Some of the edges-relationships are traversed more than the others which means those relationships occur at the result set of queries much frequently. These edges form paths which are at length from 0 to N. Collecting and analyzing these paths produces *access patterns* which are the indicators of “how the database is queried” or “how the nodes are connected from the queries’ point of view”. With the help of access patterns, paths become more important than their building blocks -the relationships.

Partitioning according to access patterns improves query performance because

Table 3.3. Relationship representation of sample graph for nodes 1, 2 and 3 in Redis.

Redis Key	Property	Value
rel:1	Start	2
rel:1	Ende	1
rel:1	Type	follows
rel:2	Start	3
rel:2	Ende	1
rel:2	Type	follows
rel:3	Start	4
rel:3	Ende	1
rel:3	Type	follows

Table 3.4. In & out relationships data structure of nodes 2 in Redis.

Redis Key	Property (Relation ID)	Value (Node ID)
in:2	7	6
in:2	8	7
in:2	9	9
out:2	1	1

related nodes and relationships based on how they are queried would be located at the same Neo4j instance. Another benefit of using access patterns would be improved load balance of the system comparing to partitioning just based on cut-edge minimization (Section 3.1.1). With *load balance* we do not mean vertex count per partition, but query count processed per partition.

As we did not have access patterns for our datasets we have generated our own access patterns by sending randomly generated queries to unpartitioned Neo4j server. For this purpose we have developed several libraries that runs on the unpartitioned Neo4j server. Below are the explanations of these libraries in detail for the graph partitioning methodologies we proposed.

3.2.2.1. APC for APWEP. The APC library that we developed for the APWEP (Section 3.1.2) case intervenes all the traversal operations performed on the Neo4j server storing the unpartitioned graph. During the traversals all visited edges are updated via increasing their weight with a predetermined value. It should be noticed that this procedure could become a performance problem in a production environment. In case of such a situation the library could be changed accordingly to perform this step in background.

Scotch library supports edge weights, so a “graph file” that considers edge weights is prepared in Scotch format as the output of this sub-step and the input of pre-partitioning phase. A sample graph file with edge weights is given in Figure 3.4.

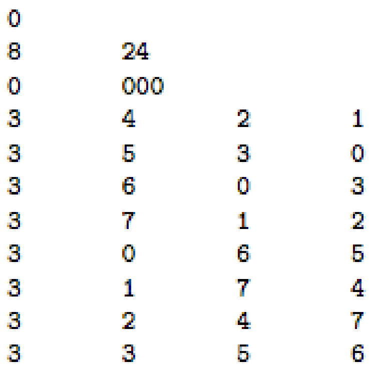
3.2.2.2. APC for APHP. APC library for the APHP case works similar to APWEP case but this time during traversals visited nodes are stored in a regular CSV file that would be the input of Preparing Neo4j Instances phase explained in Section 3.2.5.

The important part is the organization of which nodes will be written to which CSV files. The choice of the CSV file is realized by modulus operation of the start node ID for the related traversal.

There would be one less CSV files than desired partition count because of the master instance running in the system which could be thought as a partition; but in fact, it stores the original-unpartitioned graph.

3.2.3. Pre-partitioning Phase

In this phase an input file for Scotch Partitioner is prepared. The input is node IDs and relationships fetched from Redis. The output is a text file formatted as defined by Scotch and given in Figure 3.3. This file is called “graph file” and has the extension of “.grf” as a convention.



```
0
8       24
0       000
3       4       2       1
3       5       3       0
3       6       0       3
3       7       1       2
3       0       6       5
3       1       7       4
3       2       4       7
3       3       5       6
```

Figure 3.3. Sample graph file in Scotch input format with no weight.

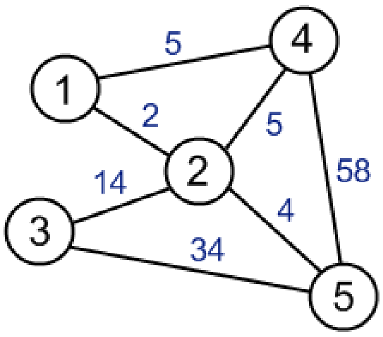
First line defines the graph file version, which is 0. The second line holds two values: Vertex number and edge number. According to the Figure 3.3 there are 8 vertices with 12 edges connecting them but edge number is written as doubled because an edge is defined in the following lines for both of its start and end vertices.

First value on the third line defines whether the index of vertices start at 0 or 1. 0 is for programming languages that starts array indexing from 0 like Java and 1 is for programming languages that starts array indexing from 1 like Fortran. The second part of the line is a numeric flag which configures the format of the following lines. A non zero value in the

- units indicates that vertex weights are provided (001)
- tenths indicates that edge weights are provided (010)
- hundredths indicates that vertex labels are provided. Providing vertex labels are useful if the vertices given in the following lines are not in natural order.

Fourth line and onwards defines the edges (relationships). The leftmost integer indicates the number of edges (edge count) that vertex has and which is unidirectional, so; both incoming and outgoing edges must be counted. In Figure 3.3 “3” means vertex with index zero has three edges. Following integers are the index numbers of connected vertices. It is strictly important that edge count and the number of neighbor vertices given must match.

In Figure 3.4 a sample *graph file* that considers edge weights is given for a weighted graph. Comparing to unweighted version first difference is in the third line, as “010” indicates that this graph file is edge weighted. And second difference is on the edge defining lines. Every edge is now defined with two numbers; first is the edge weight and second is the index number of the other node of the edge.



```
0
5   14
1   010
2   2   2   5   4
4   2   1   14 3   5   4   4   5
2   14 2   34 5
3   5   1   5   2   58 5
3   4   2   34 3   58 4
```

Figure 3.4. A sample graph with edge weights and corresponding graph file in Scotch input format.

This phase is implemented in Ruby in a serial fashion but in order to improve performance it may be implemented to work in parallel easily.

3.2.4. Partitioning

We employed Scotch library for partitioning the graph, which is an open source, graph specialized software actively developed for nearly 20 years. The input of this phase is a graph file defined in Scotch format (Section 3.2.3) and the output is a simple mapping file produced by Scotch that describes which node is mapped to which partition.

There are two configuration parameters for partitioning via Scotch: Imbalance ratio and partition count.

Imbalance ratio is the tolerance of in what percent a partition could grow or shrink in size comparing to average number of vertices. This ratio could be between 0 and 1, and passed to Scotch with *-b* parameter. Scotch uses imbalance ratio to adjust vertex count assigned to a partition. During our tests we selected imbalance ratio as 0.9 because Scotch tends to increase edge-cuts in favor of vertex count per partition. This results almost equal vertices in each partition but this time some of the highly connected clusters' members are separated which in turn causes extra traffic.

Partition count is related to domain structure. If we select a higher number of partitions than optimal, Scotch will assign members that belong to a partition to several different partitions which means extra traffic. On the other hand, if we assign lesser number of partitions than optimal, partitions will become unable to handle high load similar to unpartitioned case. During our tests we chose 10 as partition count.

3.2.5. Preparing Neo4j Instances

The input of this phase is a file that consists of mapping of nodes to partitions. This file is produced at the Partitioning Phase (Section 3.2.4) for the BP and APWEP case or at the APC Phase (Section 3.2.2.2) for APHP case. At the end of this phase Neo4j instances that are storing a partition of the whole graph are ready to serve. This phase is common to all partitioning methodologies.

According to the mapping file, 3 CSV files are generated per partition:

- Nodes CSV file
- Relationships CSV file
- Node Index CSV file

All of the CSV files are tab separated and their format is defined by Neo4j CSV Batch Importer Library [39]. Header lines of the files are the name of properties for both nodes and relationships. Following lines are the tab separated properties again both for nodes and relationships.

Batch Importer is a library developed by the main committers of Neo4j and shared under Github. The sole purpose of the library is building up the database directory of

Neo4j from the CSV files. This directory holds every single data that Neo4j needs such as nodes, relationships, node properties, relationship properties, indexes, logs and it is called “graph.db directory”.

Batch Importer uses a kernel component of Neo4j called Batch Inserter which is intended to be used for the initial import of data. In order to be fast, thread safety and transactions are sacrificed [31] within batch inserter.

After running Batch Importer with the file name arguments - nodes.csv, relationships.csv and node_index.csv - a graph.db directory is created and renamed with the ID of the partition it belongs to.

Creating the graph.db directory is followed by the transfer of these graph.db directories to S3 after being zipped in order to decrease bandwidth usage and transfer time. Within the end of transfers, launching of the EC2 instances begin. The EC2 instances boot up from pre-configured Amazon Machine Images (AMI). These images have Neo4j server installed and ready to run with all the required libraries included for connecting to Redis and result database servers.

During the boot up process of the EC2 instance, following routines are run automatically by passing a bash script to AWS:

- Graph.db directory is transferred from S3, unzipped and put under the “data” directory of Neo4j
- URL of the machine is announced via inserting the partition ID as a key and URL as a value into Redis server
- URLs of other Neo4j instances and result database, and any other configuration property is fetched from the Redis and inserted into local “interpartitiontraverse.properties” file which is accessed by Neo4j server at runtime
- A cron job [40] is added in order to fetch URLs and configuration properties from Redis within a given periodic time interval. This step is crucial to provide a failure avoidance mechanism in case of a failed and replaced Neo4j instance.

Whenever an instance is replaced its URL will be available to all other instances in a short time period.

- Finally, Neo4j server is started

3.3. Runtime Architecture

Runtime architecture consists of four main parts: Client, Redis Server, Result Database and Neo4j instances. The overall schema of the runtime architecture for BP and APWEP is shown in Figure 3.5 and APHP's runtime architecture is shown in Figure 3.6. The main difference between the two architectures are at the steps denoted as "1" and "4". Step 1 is explained in Section 3.3.2. Step 4 for the BP and APWEP case is briefly explained in Section 3.3.4. There is no step 4 for the APHP case because job delegation is not needed. A query is processed with a single instance if that query is one of the collected access patterns. Otherwise query is directed to the master instance.

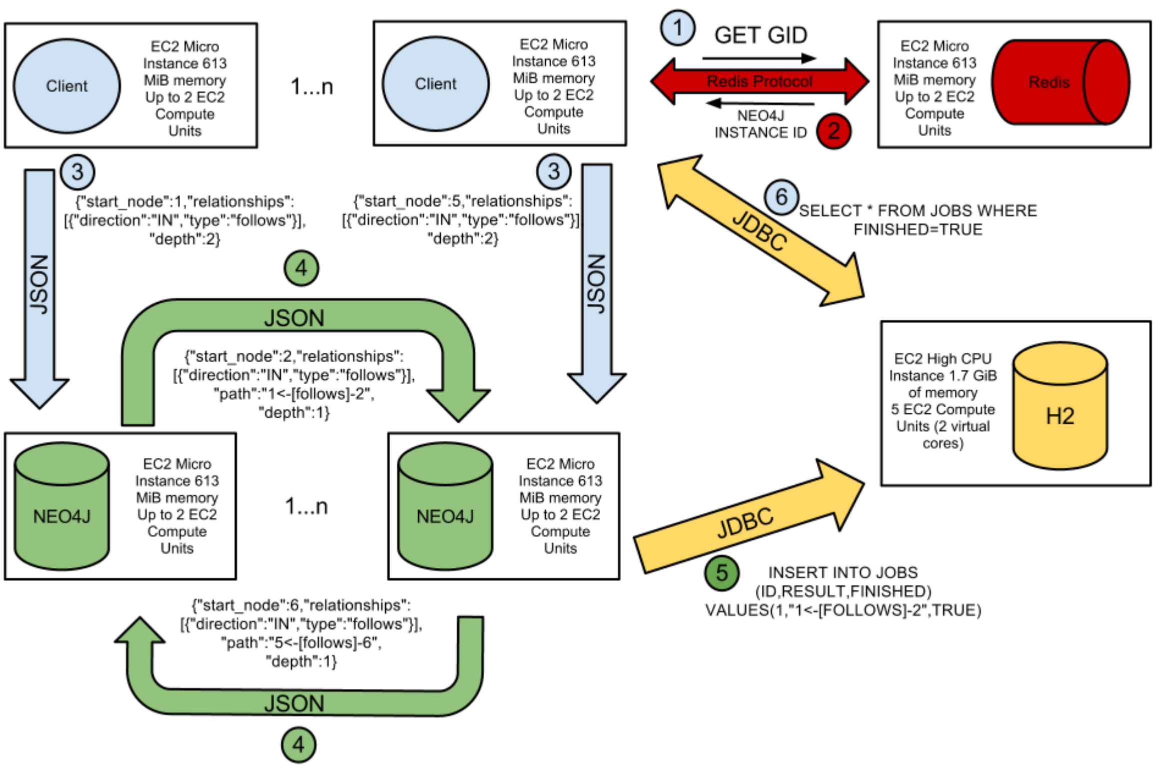

Client
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
1...n
Client
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
1
GET GID
Redis Protocol
NEO4J INSTANCE ID
2
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
Redis
3
{"start_node":1,"relationships": [{"direction":"IN","type":"follows"}], "depth":2}
3
{"start_node":5,"relationships": [{"direction":"IN","type":"follows"}] "depth":2}
JSON
JSON
6
SELECT * FROM JOBS WHERE FINISHED=TRUE
JDBC
4
JSON
{"start_node":2,"relationships": [{"direction":"IN","type":"follows"}], "path":"1<-[follows]-2", "depth":1}
EC2 High CPU Instance 1.7 GiB of memory 5 EC2 Compute Units (2 virtual cores)
H2
NEO4J
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
1...n
NEO4J
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
JDBC
5
INSERT INTO JOBS (ID,RESULT,FINISHED) VALUES(1,"1<-[FOLLOWS]-2",TRUE)
{"start_node":6,"relationships": [{"direction":"IN","type":"follows"}], "path":"5<-[follows]-6", "depth":1}
JSON
4

Figure 3.5. Runtime architecture of BP and APWEP.

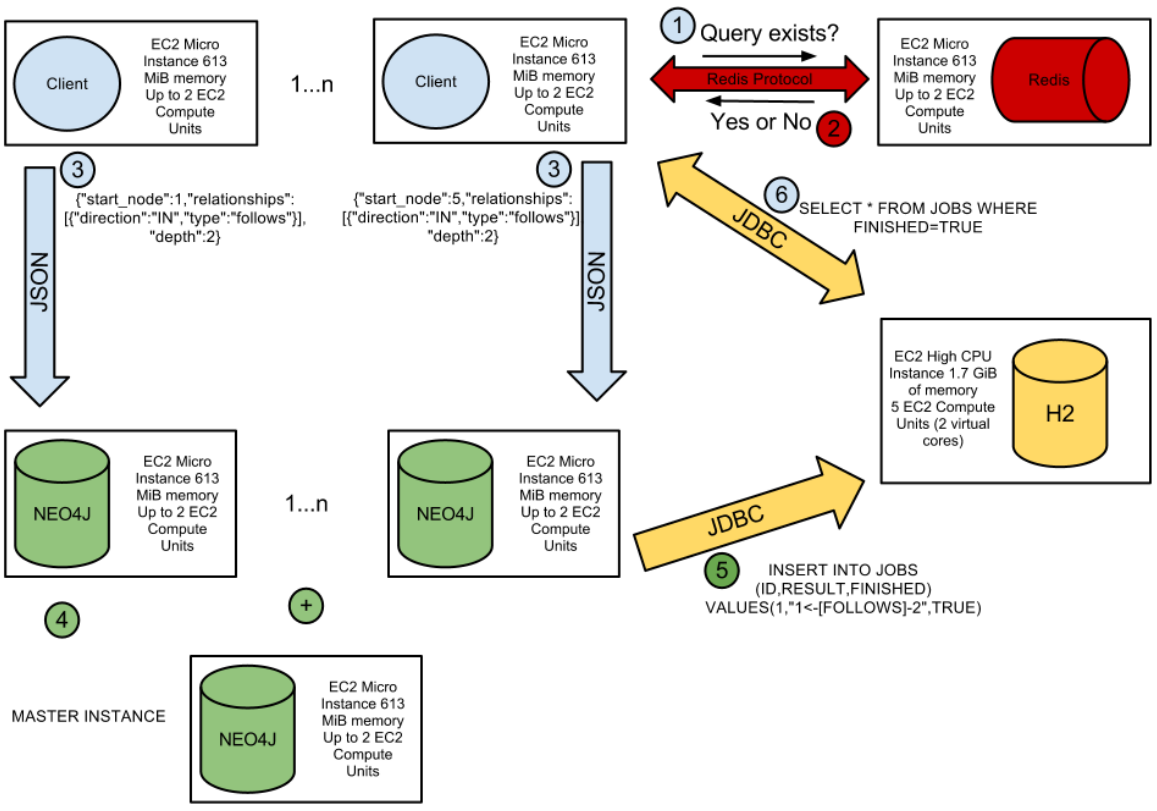

Client
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
1...n
Client
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
1 Query exists?
Redis Protocol
Yes or No 2
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
Redis
3
{"start_node":1,"relationships":[{"direction":"IN","type":"follows"}],"depth":2}
3
{"start_node":5,"relationships":[{"direction":"IN","type":"follows"}],"depth":2}
JSON
JSON
6 SELECT * FROM JOBS WHERE FINISHED=TRUE
JDBC
EC2 High CPU Instance 1.7 GiB of memory 5 EC2 Compute Units (2 virtual cores)
H2
NEO4J
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
1...n
NEO4J
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units
JDBC
5 INSERT INTO JOBS (ID,RESULT,FINISHED) VALUES(1,"1<-[FOLLOWS]-2",TRUE)
4
+
MASTER INSTANCE
NEO4J
EC2 Micro Instance 613 MiB memory Up to 2 EC2 Compute Units

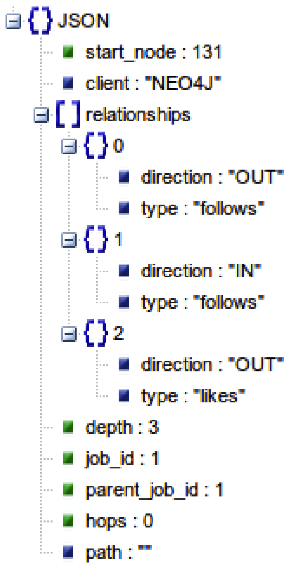
Figure 3.6. Runtime architecture of APHP.

3.3.1. Client

Client is the owner of a graph query. It could be a web browser operated by an end-user or it could be a server sitting in between our server and a web browser for several reasons such as security or caching.

Java is chosen for implementing the client but any other programming language that supports sending JavaScript Object Notation (JSON) over REST is applicable. A sample JSON sent from a client to the system is given in Figure 3.7. The query is "Give me the nodes that are liked by people following the people that I follow" which could be used as a recommendation query as it may fetch liked objects by people that have similar tastes as mine. Definition of the key-value pairs in JSON is as follows:

- Traversal should begin at the node with GID 131. Local ID (Neo4j given ID) may differ from GID
- Three relationships should be traversed: First, an outgoing relationship with “follows” type. Second, an incoming relationship with “follows” type. And third, an outgoing relationship with “likes” type
- This the first job created for this query, so job_id and parent_job_id fields are the same and given as 1. This ID is the ID generated at H2 Database auto incrementally
- Hop count, denoted as hops, is given as 0 as this is the initial query and no hop occurred before
- Previous path, denoted as path, is also empty as this is the initial query



```
{} JSON
   start_node : 131
   client : "NEO4J"
   [] relationships
      {} 0
         direction : "OUT"
         type : "follows"
      {} 1
         direction : "IN"
         type : "follows"
      {} 2
         direction : "OUT"
         type : "likes"
   depth : 3
   job_id : 1
   parent_job_id : 1
   hops : 0
   path : ""
```

Figure 3.7. JSON sent from client to the system in order to start a query.

3.3.2. Redis Server

We used Redis both in Partitioning and Runtime Architecture parts. Former one is explained in Section 3.2.1 with the reason behind choosing Redis.

Redis is the first connection point of our system from the client’s point of view. Clients make use of Redis to find out in which partition the real starting node of a query exists, so the query could be directed to that instance.

For the BP and APWEP case Redis stores mapping of GIDs to partitions. So, before submitting a job a client first connects to the Redis and fetches the partition ID of the starting node, then query is submitted to the Neo4j with that ID.

On the other hand, finding out the partition that stores the starting node is performed by a modulo calculation for the APHP case, so there is no need for a lookup operation from Redis. This time, Redis is used for checking whether the query is one of the access patterns collected before. If so, query is sent to the related partition, otherwise sent to master instance. This procedure is given in Figure 3.2.

3.3.3. Result Database

In our distributed architecture, results are collected in a relational database that runs on a separate machine independent from clients or Neo4j instances. We have chosen to use a relational database as the result of our queries were suitable to be stored in tabular format. H2 [41] was the selected database management system because of its adequate performance coming with a small footprint and because of its being open source, free software.

Design of the database is kept at minimum and a single table with the name "NEORESULTS" is used. The table references itself on the ID column with PARENT_ID column but without a foreign key constraint in order not to sacrifice performance. ID column could be thought as the ID of the job or query submitted to the system in order to be executed by Neo4j instances and it is generated by the database management system auto incrementally. Every query submitted to the system creates an initial job which is the parent of other jobs that may be created by this query if there happens a hop to other Neo4j instance during the traversal. At this point, PARENT_ID column is used to keep track of every subquery created with the parent - initial - job. Only the first job's PARENT_ID is equal to its ID.

Parent jobs are submitted by clients but sub-query jobs are submitted by Neo4j instances. So, in either case the responsibility of updating and setting the PARENT_ID of a job belongs to the submitter of it.

VQUERY column holds the string representation of the query in JSON format for further analysis. A caching layer at the front of the system may use this column in order to skip queries performed in recent times defined by a policy.

PARTITION column is used just for reporting how well the vertices are scattered to partitions. At the beginning of this query, after the decision of target partition via fetching from Redis or modulus operation, the target partition number is written in this column. The more a partition number occurring in the table means the more it is used for traversals.

VRESULT column is used to hold results of queries or part of them if it is a subquery. The column is filled by the Neo4j instance that fulfilled related query.

IS_DELETED is a boolean column that provides a softdelete mechanism. At the beginning, when the job is submitted it is marked as FALSE. Whenever a client fetches the result of that job it is marked as TRUE which indicates result is fetched by a client and will not occur in the unfinished jobs list from then on; as if the row is deleted.

CREATED_AT is a timestamp column that holds the creation time of a query. It is filled by the client or by the Neo4j instance that is forwarding rest of its traversal query to another Neo4j instance.

FINISHED_AT is a timestamp column that holds the time given query in VQUERY column is finished. It is filled by the Neo4j instance performed the traversal.

A row in the NEO4JRESULTS table corresponds to a job or query and has 3 states:

- NEW: At this stage job is submitted with a PARENT_ID and the query is about to be run or running on a Neo4j instance. VRESULT is NULL and IS_DELETED is FALSE
- COMPLETED: This state means that query is fulfilled and waiting for a client read it. VRESULT is filled with the query result and IS_DELETED is FALSE
- DELETED: This state means that completed query is also read by the client. IS_DELETED is TRUE

3.3.4. Neo4j Instances

Neo4j instances are the most important part of the Runtime Architecture that provides parallel execution of queries and high throughput with horizontal scaling.

Neo4j instances perform two operations: Fulfilling a query or delegating part of the query. In order to support these operations standard Neo4j server is extended within the context of our thesis.

Neo4j provides a REST endpoint with the purpose of accepting connections from clients. We have extended this mechanism via developing a plugin which accepts queries defined in JSON and operates on the graph database running under the hood. The plugin also communicates with the result database for sending query results.

At the first step, JSON sent over REST is parsed and converted into a traversal description in Java language that works with Neo4j Graph Database API directly. And then the query is executed as a traversal.

Traversing operation, which depends on “shadow nodes”, is the key point of our architecture. Shadow nodes are the connecting point of the partitions. They are created after the partitioning process. During the creation of relationships of the real nodes in a partition, whenever other node of the relationship does not exist in that partition; a shadow node is created and linked with the relationship. A sample graph with 11 nodes is given in Figure 3.8 as before partitioning and in Figure 3.9 as after partitioning. Gray colored nodes denote shadow nodes and dashed lines are used just to point shadow nodes' real counterparts. They do not exist as an edge in real graph.

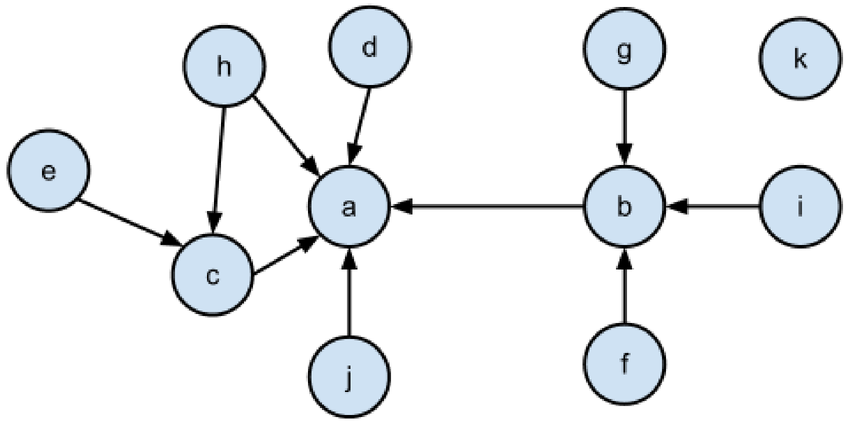


Figure 3.8. Unpartitioned sample graph with 11 nodes.

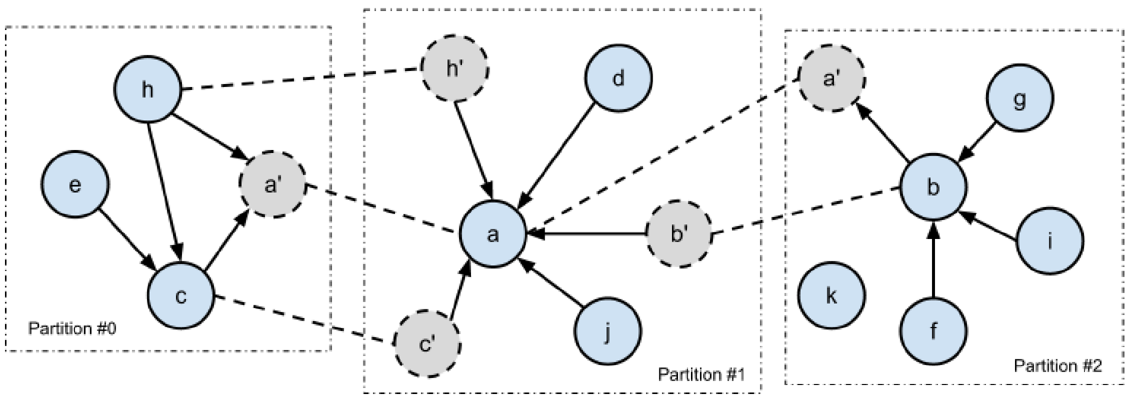


Figure 3.9. Partitioned sample graph with 11 nodes.

Shadow nodes hold exactly the same properties that their real counterparts hold. Their relationships to real nodes also have the same properties as the original graph before partitioning but shadow nodes may not have all the relationships that their real counterparts have. Because; relationships are created only according to real nodes and shadow nodes are complementary. There is only one shadow node corresponding to a real node in a given partition but there may be other shadow nodes in other partitions.

Our solution is different from the one proposed in [30] as there is no distinction between relationships, which means all the relationships are real. A relationship between a real node and a shadow node is the reflection of the original relationship and it exists in at least two partitions.

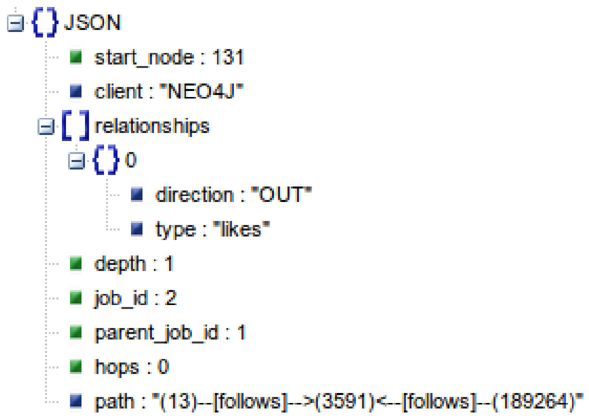
Comparing to [30] another difference is related to globally unique ID called as GID. GID is the ID assigned to nodes by Neo4j before partitioning. After partitioning, every node is assigned two properties -GID and shadow- whether it is a real node or a shadow node. GID value is used as the key while the node is being added to the *nodes index* within a partition. Obviously, *Shadow* property is used to discriminate shadow nodes and has the boolean value TRUE or FALSE.

Our plugin decides when to continue the execution of the query on a different partition based on shadow property. During a traversal every node visited is tested against their shadow property and when the property is TRUE the rest of the query is delegated to related partition for execution. There is an exception for this flow: If the last visited node is a shadow node but at the same time traverser is at the max depth specified by the query, which indicates that the traverse should end, then no delegation occurs as the query is performed completely.

Query delegation process is similar to query submitting performed by the clients; a job is created in the result database and then a JSON defining the query is prepared and submitted to another Neo4j instance that is holding the related partition. The difference is in the JSON preparation process which is the modification of the original JSON received in 6 steps:

- The job ID is modified by the new job ID
- The relationships defined in the original JSON is pruned up to the depth that the traverser stopped
- Initial depth is decreased by already traversed depth
- Start node GID is updated with the shadow node's GID
- The path up to this point is updated
- Hop count (number of additional machines involved) is increased by 1

Relationships, depth and start node GID is related to traversal but job ID will be used to update Result Database, path will be the part of the query result and hop count is just for information. A sample JSON sent during delegation is given in Figure 3.10. It may easily grasped from the sample that:



```
{} JSON
   start_node : 131
   client : "NEO4J"
   [ ] relationships
      {} 0
         direction : "OUT"
         type : "likes"
   depth : 1
   job_id : 2
   parent_job_id : 1
   hops : 0
   path : "(13)--[follows]-->(3591)<--[follows]--(189264)"
```

Figure 3.10. Sample JSON sent during delegation.

Query delegation is an asynchronous operation performed in a background thread and the traversal operation goes on with other nodes and paths while delegations are performing. This is one of the reasons behind the impossibility of guessing the number of jobs that will be created during any query. Another reason is that a Neo4j instance is not able to guess how many hops will be needed to perform a query if it is not the only instance performing the query.

The Result Database is updated with finished path or paths when the traversal comes to an end. A finished path is a path that has no shadow node except the last node. Paths may have been traversed by the Neo4j instance itself or parts of it may be collected from other instances and accumulated with the paths from this instance.

4. TESTS AND EVALUATION

4.1. Datasets

We have used two real world datasets. First one is a web graph that is provided by Erdös Webgraph Project [2] developed by Institute of Mathematics, Eotvos University. And the second one is Pokec social network dataset collected by Lubas Takac & Michal Zabovsky [42] and shared under Stanford Network Analysis Project [43].

4.1.1. Erdös Webgraph Dataset

Erdös Webgraph dataset is a subset of World Wide Web. It is updated weekly and allowed to be used in research projects freely. Dataset is given in a 2 column format. First column denotes the source domain-name and the second column denotes target domain-name, so those are the vertices. And as a whole, a line reflects a link, which is a directed edge, from the source domain-name to the target domain-name. There is a link between two domain-names if there exists a document under the source domain-name and targets a document under the target domain-name. Domain-names are encoded in 26 character long strings such as “01324moja6i5ghdbhfe94iou9e” which could be resolved to a domain-name by a service provided by Erdös Webgraph Project.

The graph uses domain-names instead of URLs because URLs are grouped under domain-names. For example boun.edu.tr is a domain-name and there are URLs such as http://boun.edu.tr/History.aspx targeting a document or http://boun.edu.tr/Assets/Images/banner.jpg targeting a picture under the sub-directories of the same domain--name. These sub-directories are not treated as distinct vertices and they are all accepted the same as http://boun.edu.tr vertex. But a sub-domain such as http://cmpe.boun.edu.tr is a distinct domain-name and treated as a new vertex.

The size of the dataset is 1.8M vertices and 16M edges. Erdös Webgraph Dataset resembles large online social networks structure as the degree distribution fits the Power

Law and shown in Figure 4.1.

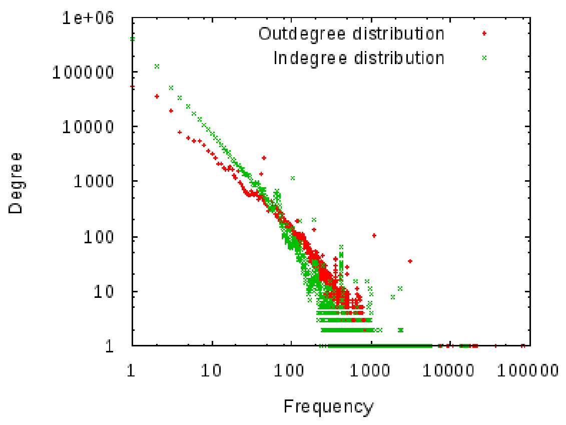


Figure 4.1. Degree distribution of Erdös Webgraph Dataset [2].

4.1.2. Pokec Social Network Dataset

Pokec is the biggest online social network in Slovakia and more popular than even Facebook with a history of 10 years. The dataset holds 1.637.068 nodes and 46.171.896 relationships. Similar to Erdös Webgraph dataset, the degree distribution of the Pokec dataset fits in a power law distribution as stated in the work of Takac & Zabovsky [42]. Besides the degree distribution another typical property of scale-free networks is also observed in the dataset, which is the average distance by nodes should lie between the values given as

$$\ln \ln N < \mathit{diameter} < \ln N / \ln \ln N \tag{4.1}$$

and when N is replaced by the total number of nodes, 1.6M, the result confirms that the dataset is a scale-free network:

$$2.66 < 4.67 < 5.38 \tag{4.2}$$

The dataset contains the profile data of users but during our tests we only used the relationships between users which are given in a CSV file format with two columns, each holding the ID values of nodes of a relationship.

4.2. Tests

4.2.1. Query (Traversal) Performance Testing

Testing of traversal performance involves two metrics: Query duration and extra traffic.

Query duration is the time difference between a client submits a query and gets the results. Comparing to extra traffic, query duration is harder to measure as it is time dependent and time depends on semi-controlled environmental variables such as network delays and resource (CPU) scheduling of virtual environments such as AWS.

As we were not able to collect access patterns of the queries over selected datasets we generated random access patterns. Then we recorded these random access patterns and applied the same set of access patterns over the same data partitioned with three graph partitioning methodologies we proposed.

Extra traffic is the traffic generated by Neo4j instances delegating rest of their jobs to other Neo4j instances. It is simply calculated by counting the sub-queries submitted under the same query which means that have the same parent job id referenced. Lower values are better as in the best case scenario all the related nodes and relationships to the query are located at the same Neo4j instance which means no extra job is submitted except the first one sent to the system. Overall aim of the system is decreasing the number of extra traffic per query which depends on the queries, partitioning quality and the underlying graph data structure. The queries are not under control of our framework but they affect the partitioning so they are implicitly related to extra traffic.

Extra traffic tests are only performed on partitioned instances as there would always be just one job for a query over an unpartitioned database. Similarly, APHP is not included in these tests as there is no job delegation, so no extra traffic.

Figures 4.2, 4.3 and 4.4, 4.5 displays the query performance results of 2 and 3 depth queries of BP and APWEP methodologies. Respond duration of a query is given with respect to hop counts which means extra traffic generated. "0" values on the $x-axis$ means no traffic generated for that query as the traversal is performed by a single Neo4j instance.

Tests are performed for two cases: Cold cache and warm cache. Cold cache is the situation that a query is sent to the system when node-relationship caches of the partitions do not have related nodes or relationships; so, a disk read operation, which increases the time spent for a query, occurs. From then on, related nodes and relationships for the same query are served from the cache which brings us to the warm cache situation. Cold cache values on the figures correspond to the first time that query is sent to the system and warm cache values are the average results of running the same query for several times. It may easily be observed from the figures that warm cache outperforms cold cache as expected but cache is a limited resource and gets full with different values related to different queries. Caching is a key topic for Neo4j queries and explained in the next section, Section 4.2.2.

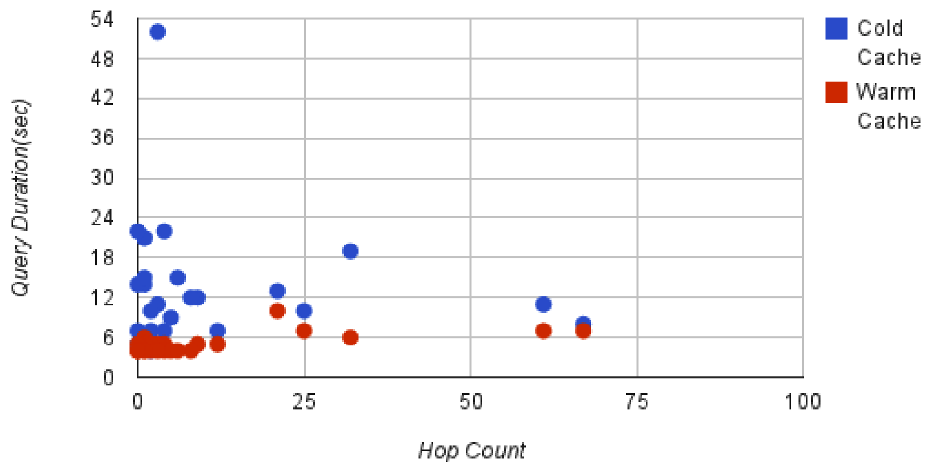


Figure 4.2. BP 2 depth query performance results with extra traffic.

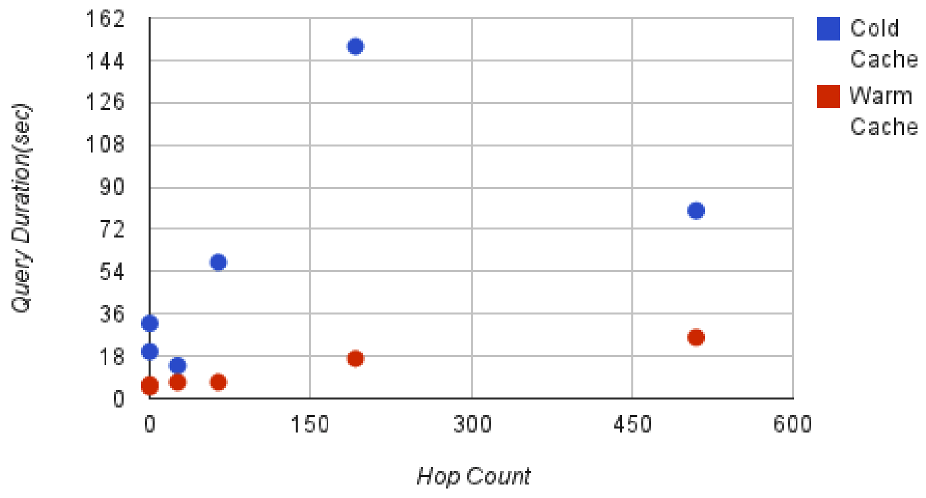


Figure 4.3. BP 3 depth query performance results with extra traffic.

Results of the query performance tests show that APWEP is good for both 2 and 3 depth queries as expected. BP is also in the acceptable range with some outlier values.

4.2.2. Load Testing

One of the aims of our framework was increasing throughput of the system via distributing load over several Neo4j instances. Neo4j graph database uses Java under the hood, so all the entities (nodes, relationships, properties, indexes) are kept as Java objects during their lifetime in memory. Different queries means creating new objects via loading from disk to Java Runtime Environment or destructing them when they are out of scope. These destructed objects are collected by the Java Virtual Machine's (JVM) Garbage Collector (GC) from time to time in order to free up the memory hold by unreferenced objects. Garbage collection could become one of the bottlenecks of Neo4j server which is a CPU intensive operation. It causes the server become unresponsive which is called as *GC Pause*.

The other bottleneck is disk reads, that's why Neo4j utilizes different levels of caches exhaustively. Node-relationship cache is located in the Java heap and used as

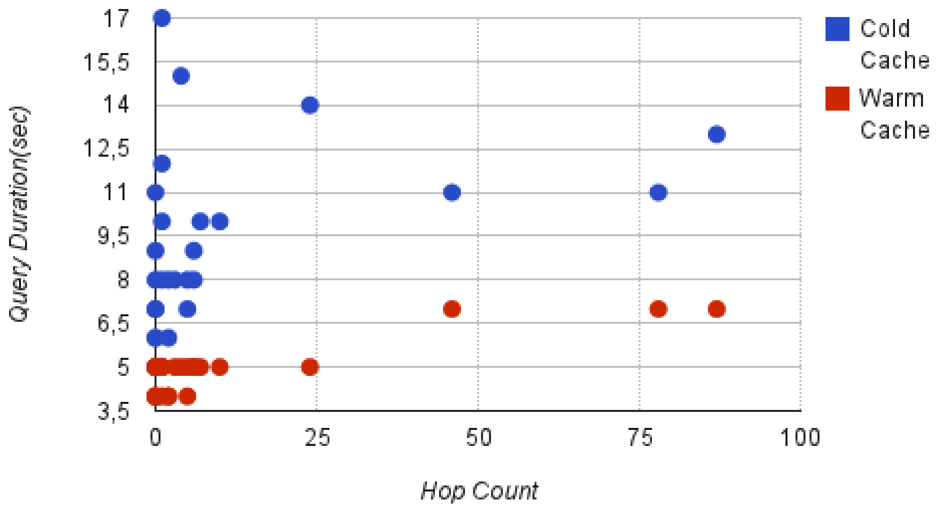


Figure 4.4. APWEP 2 depth query performance results with extra traffic.

first level cache; file cache is used for memory mapping as second level cache.

Neo4j uses memory mapped files for input-output operations. In the best case all the nodes, relationships and their properties are mapped in memory but this is not feasible for most of the real world scenarios so Neo4j does its best to use the available memory. The memory used for mapping is allocated outside of the heap area used by JVM and Neo4j uses this mapped memory as a disk cache. Whenever a miss occurs in node-relationship caches, disc cache is used. Neo4j tries to update its memory mapping via swapping with the most recently values so disc read is indispensable. During our tests and analysis, disc read operations were the root cause of long running queries. This may be seen easily from the Figures 4.2, 4.3, 4.4 and 4.5 as there are substantial differences between the duration of queries when the cache is warm and cold.

Load testing measures up to what point our system keeps on responding clients under heavy load. We tested 2 and 3 depth queries for load testing. The selection of the random queries are based on same access patterns that were used for partitioning the graph. A specialized client that produces high volume of queries, keeps track of query results and warns if a server does not respond is also developed for the purpose.

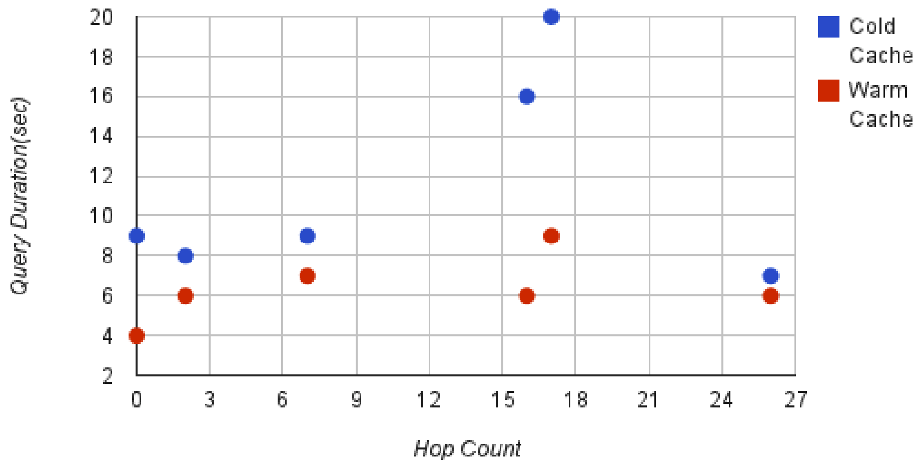


Figure 4.5. APWEP 3 depth query performance results with extra traffic.

Figure 4.6 and 4.7 show the results of our tests for 2 and 3 depth queries respectively and again lower values are better. The most important point that we need to emphasize is *unpartitioned Neo4j instance*'s results are only shown for 5 and 10 "2 depth" concurrent queries as over that load it becomes unresponsive or responds in several minutes (up to half an hour) instead of seconds which is unacceptable for a real world environment. All partitioning methodologies improve load handling of the overall system as they stay responsive for higher loads. APHP makes the best scores for both 2 and 3 depth queries as expected. APWEP also stays in the acceptable part of the results. BP performed similarly to APWEP and outperformed unpartitioned instance. In our opinion, weight increase policy that we have selected has a huge impact on the results. A deeper analysis of weight assignment may cause different results with different weight assignment policies.

Load distribution to each partition for APHP, BP and APWEP partitioning methodologies are given in Figures 4.8, 4.9 and 4.10 respectively. The numbers are the partition numbers and percentages show their involvement in processing of queries. Involvement of a partition means having whole traversal or just part of it realized by that partition.

The load distribution data given in the figures supports the load test results given

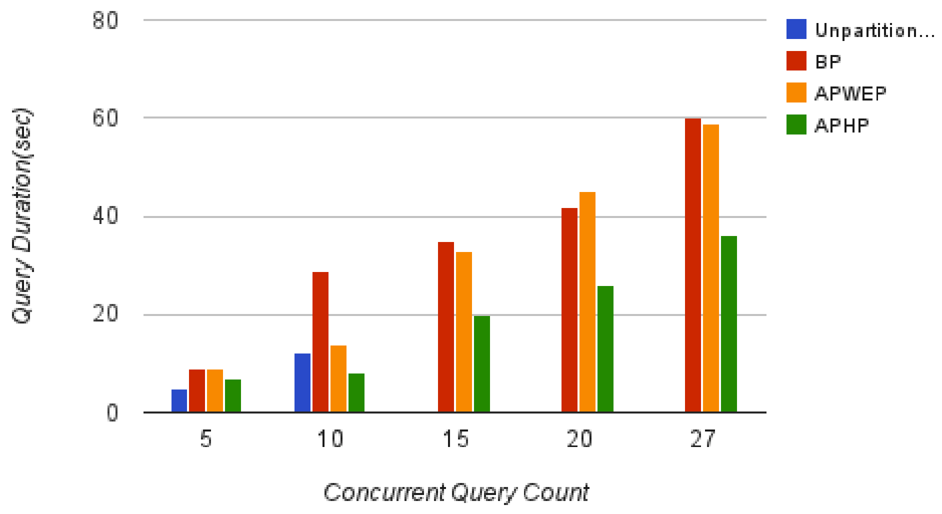


Figure 4.6. 2 depth load test results.

before. Having a better, near to optimal distributed load balance, APHP responds to queries in a shorter time period than other methodologies. BP and APWEP have some unbalanced partitions which may become a hot spot for some access patterns; as a result both of them respond to concurrent queries within a time period close to each other and stayed in the acceptable range.

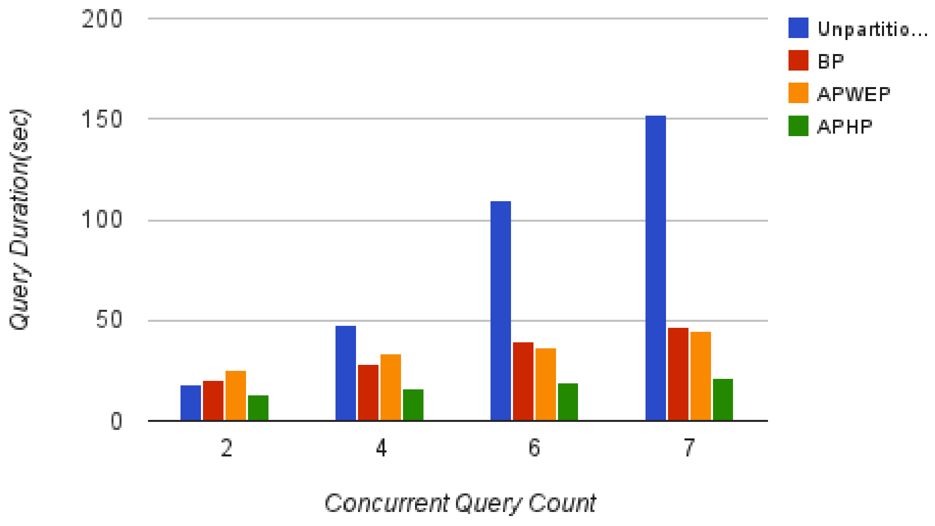


Figure 4.7. 3 depth load test results.

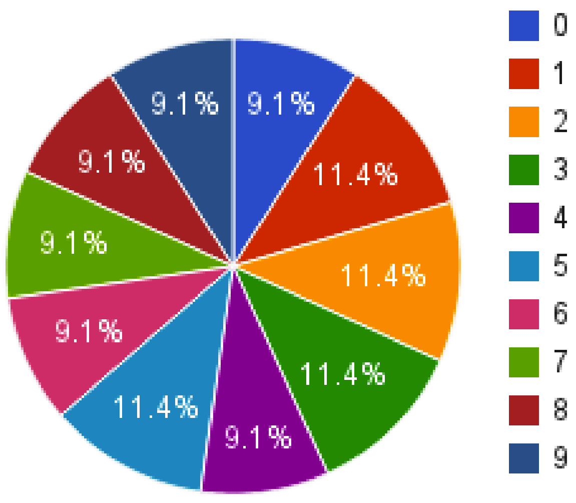


Figure 4.8. APHP load distribution.

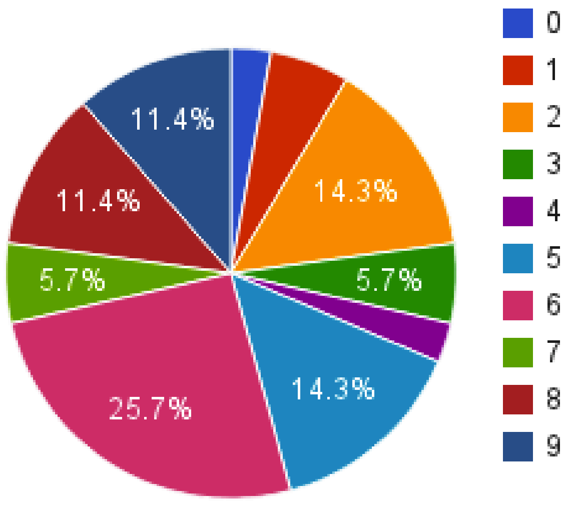


Figure 4.9. BP load distribution.

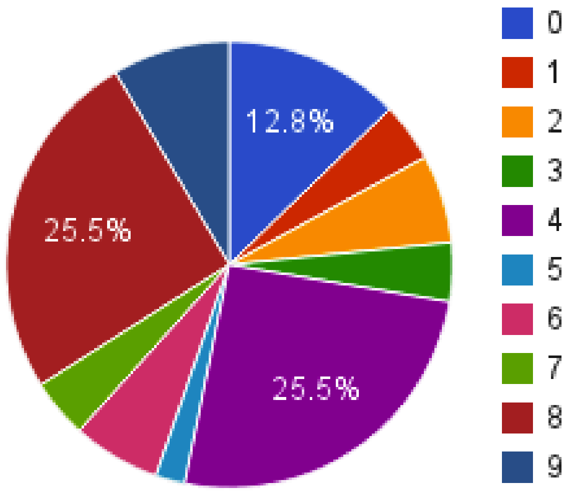


Figure 4.10. APWEP load distribution.

5. CONCLUSION

Our evaluations showed that horizontally scaling via graph partitioning could improve the query performance and load handling capacity of a system. Despite being an NP-hard problem, graph partitioning could be alleviated by applying access patterns as an heuristic. Even uninformed partitioning could achieve high throughput ratios comparing to an unpartitioned instance. Additionally we showed that access patterns could be used to collect sub-graphs in order to form a perfectly balanced cache solution.

This kind of a framework could be employed in situations that suffer high load on a single instance. To the best of our knowledge there is no such a system implementation makes use of access patterns.

Our framework was developed with some assumptions made such as lack of failover mechanism or update propagation through the instances but we think that these problems are solved in other distributed solutions and integrating them with our system would be a matter of time.

5.1. Contribution

In this work we dived into the world of scaling graph databases via partitioning. For this purpose we proposed and developed a graph partitioning framework and a distributed graph database architecture as well. We have performed tests comparing three partitioning methodologies with query performance, extra traffic generated and load handling aspects. We have used two real world dataset, one from a social network and the other from a web graph, with substantial amount of nodes and relationships (1.8M, 1.6M nodes and 18M, 46M relationships).

5.2. Future Work

In our work we used real world datasets but our access patterns were randomly generated. It is required to repeat test cases with real access patterns collected from a running system.

Insert time partitioning could be applied to improve the quality of the partitions after partitioning. Similarly local partitioning are also required to have a sustainable system as our partitioning methodologies require accessing whole graph.

Decreasing the extra traffic via intervening job delegations between instances could be an extension to our work. With such a mechanism different jobs delegated to an instance could be combined into coarser jobs coming from the same origin.

Lastly, our JSON query parser could be improved to support more constraints related to property values of nodes and relationships.

REFERENCES

1. Neubauer, P., "Graph Databases, NOSQL and Neo4j", 2010, http://www.infoq.com/articles/graph-nosql-neo4j, accessed at April 2013.

2. Eötvös University, "Erdös Webgraph Project", 2010, http://web-graph.org/, accessed at April 2013.

3. Borthakur, D., "Facebook has the world's largest hadoop cluster", 2010, http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html, accessed at April 2013.

4. Averbuch, A. and M. Neumann, *Partitioning Graph Databases*, Master's thesis, KTH Computer Science And Communication, 2010.

5. Amazon.com Inc., "Amazon Web Services", 2006, https://aws.amazon.com/, accessed at April 2013.

6. Strozzi, C., "NoSQL, A Relational Database Management System", 1998, http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page, accessed at April 2013.

7. Oskarsson, J., "NoSQL 2009", 2009, http://blog.sym-link.com/2009/05/12/nosql_2009.html, accessed at April 2013.

8. Lo, C. Y. Z., Y. He, and C. P. Lin, "Graph theoretical analysis of human brain structural networks", *Reviews in the Neurosciences*, Vol. 22, No. 5, pp. 551–563, 2011.

9. Rodriguez, M. A. and P. Neubauer, "Constructions from dots and lines", *Bulletin of the American Society for Information Science and Technology*, Vol. 36, No. 6, pp. 35–41, 2010.

10. Free Software Foundation Inc., "GNU Affero General Public License", 2007, `http://www.gnu.org/licenses/agpl-3.0.html`, accessed at April 2013.

11. Mondal, J. and A. Deshpande, "Managing large dynamic graphs efficiently", *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 145–156, ACM, New York, NY, USA, 2012.

12. "Object-relational impedance mismatch", 2005, `http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch`, accessed at April 2013.

13. Facebook Inc., "Introducing Graph Search", 2013, `https://www.facebook.com/about/graphsearch`, accessed at May 2013.

14. Robinson, I., J. Webber, and E. Eifrem, *Graph Databases*, O'reilly, Sebastopol, 2013.

15. Era7 Bioinformatics, "Bio4j, bio data graph db", 2012, `http://bio4j.com/`, accessed at April 2013.

16. Partner, J., A. Vukotic, and N. Watt, *Neo4j In Action*, Manning, New York, 2013.

17. Neo Technology, "Cypher query language", 2012, `http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html`, accessed at April 2013.

18. Tinkerpop, "Gremlin, graph traversal language", 2012, `https://github.com/tinkerpop/gremlin/wiki`, accessed at April 2013.

19. "Groovy, a dynamic language for the Java platform", 2007, `http://groovy.codehaus.org/`, accessed at April 2013.

20. Mernik, M., J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages", *ACM Computing Survey*, Vol. 37, No. 4, pp. 316–344, December 2005.

21. Barabási, A. L. and R. Albert, "Emergence of Scaling in Random Networks", *Science*, Vol. 286, No. 5439, pp. 509–512, October 1999.

22. Adamic, L. A. and B. A. Huberman, "Zipf's law and the Internet", *Glottometrics*, Vol. 3, pp. 143–150, 2002.

23. Thomae, O. R. M., *Database Partitioning Strategies for Social Network Data*, Master's thesis, Massachusetts Institute of Technology, 2012.

24. Donath, W. E. and A. J. Hoffman, "Lower bounds for the partitioning of graphs", *IBM Journal of Research and Development*, Vol. 17, No. 5, pp. 420–425, September 1973.

25. Pratt, T. W. and D. P. Friedman, "A language extension for graph processing and its formal semantics", *Commun. ACM*, Vol. 14, No. 7, pp. 460–467, July 1971.

26. Angles, R. and C. Gutierrez, "Survey of graph database models", *ACM Computing Survey*, Vol. 40, No. 1, pp. 1:1–1:39, February 2008.

27. Angles, R., "A Comparison of Current Graph Database Models", *ICDE Workshops*, pp. 171–177, 2012.

28. Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing", *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pp. 135–146, ACM, New York, NY, USA, 2010.

29. Valiant, L. G., "A bridging model for parallel computation", *Communications of the ACM*, Vol. 33, No. 8, pp. 103–111, August 1990.

30. Chairunnanda, P., S. Forsyth, and K. Daudjee, "Graph data partition models for online social networks", *Proceedings of the 23rd ACM conference on Hypertext and social media*, HT '12, pp. 175–180, ACM, New York, NY, USA, 2012.

31. Neo Technology, “Memory mapped IO settings”, 2012, `http://docs.neo4j.org/chunked/milestone/configuration-io-examples.html#configuration-batchinsert`, accessed at April 2013.

32. Neo Technology, “Neo4j relationships”, 2012, `http://docs.neo4j.org/chunked/stable/graphdb-neo4j-relationships.html`, accessed at April 2013.

33. Ho, L.-Y., J.-J. Wu, and P. Liu, “Distributed Graph Database for Large-Scale Social Computing”, *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 455 –462, June 2012.

34. “GoldenOrb”, 2013, `http://goldenorbos.org/`, accessed at April 2013.

35. Karypis, G. and V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”, *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, pp. 359–392, 1998.

36. “Graph Partitioner”, 2012, `https://github.com/volkantufekci/Neographic`, accessed at April 2013.

37. “Distributed graph database framework”, 2013, `https://github.com/volkantufekci/Neo4jCSVParserHubway`, accessed at April 2013.

38. Pellegrini, F., J. Roman, H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, “Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs”, *High-Performance Computing and Networking*, Vol. 1067, pp. 493–498, Springer Berlin Heidelberg, 1996.

39. Hunger, M., “Generic csv file neo4j batch importer”, 2012, `https://github.com/jexp/batch-import`, accessed at April 2013.

40. Vixie, P., “Cron Job”, 1993, `http://unixhelp.ed.ac.uk/CGI/man-cgi?crontab+5`, accessed at April 2013.

41. Mueller, T., “H2 Database Engine”, 2005, `http://www.h2database.com/html/main.html`, accessed at April 2013.

42. Takac, L. and M. Zabovsky, “Data Analysis in Public Social Networks”, *International Scientific Conference and International Workshop Present Day Trends of Innovations*, May 2012.

43. Leskovec, J., “Stanford Network Analysis Project”, 2009, `http://snap.stanford.edu/`, accessed at April 2013.